İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

PROLOG'UN PARALEL MANTIK PROGRAMLAMAYA GENİŞLETİLMESİ

DOKTORA TEZİ

Y. Müh. Nazım KOÇ
(511942036)

Tezin Enstitüye Verildiği Tarih : 24 Ocak 2003
Tezin Savunulduğu Tarih : 28 Temmuz 2003

Tez Danışmanı :	**Doç.Dr. Şakir KOCABAŞ**
Diğer Jüri Üyeleri	**Prof.Dr. Ertuğrul TAÇGIN (M.Ü.)**
	Prof.Dr. Ercan ÖZTEMEL (S.Ü.)
	Doç.Dr. Coşkun SÖNMEZ (İ.T.Ü.)
	Doç.Dr. Selamet ERÇELEBİ (İ.T.Ü.)

TEMMUZ 2003

ÖNSÖZ

Son derece sade ve etkileyici bir programlama dili olan Prolog üniversitelerimizde hakettiği yeri alamamış ve yeterli ilgiye mazhar olamamıştır.

Prolog konusunda olduğu gibi Paralel Prolog konusunda da çok az çalışma yapılmıştır. Yüksek lisans tez tarama sistemi ile yaptığımız bir araştırmada, 1984'ten günümüze kadar Prolog konusunda 16 adet tez yapılmış olduğunu gördük. Bunlardan sadece iki tanesi doktora tezidir. Paralel Prolog konusunda ise doktora seviyesinde herhangi bir araştırmaya rastlayamadık.

Yapmış olduğumuz bu doktora tezi çalışmasında standard bir sıralı Prolog derleyicisini alarak paralel bir sistem haline dönüştürdük. Yeni bir paralel mantık programlama dili önerdiğimiz gibi, bu dilin TCP/IP ortamında işleyen bir yürütme modelini de gerçekleştirdik.

Bu tezin tamamlanmasında ve bütün doktora çalışması sırasında bana büyük bir sabır ile destek veren değerli hocam Doç. Dr. Şakir Kocabaş'a ve bütün jüri üyelerine teşekkür ederim.

OCAK 2003 Nazım KOÇ

İÇİNDEKİLER

ÖNSÖZ ii
KISALTMALAR vi
TABLO LİSTESİ vii
ŞEKİL LİSTESİ viii
SUMMARY ix
ÖZET x

1. GİRİŞ 1

2. PARALEL MANTIK PROGRAMLAMA 3
2.1. Sıralı Prolog 3
2.2. Warren'in Soyut Makinesi 4
2.3. Paralel Prolog 4
2.4. Örtük Paralellik 5
2.5. Bağımlı-Ve Paralelliğin Uygulanmasındaki En Genel Problem 9
2.6. Paralel Mantık Programlamada Yaklaşımlar 9
2.7. CCND Dilleri 10
2.7.1. Ortak Özellikler 12
2.7.2. Klasik CCND Dilleri 14
2.7.2.1. Parlog 15
2.7.2.2. Concurrent Prolog 16
2.7.2.3. Guarded Horn Clause 18
2.7.3. Uygulama Modelleri 19
2.7.3.1. Multi-PSI Modeli 19
2.7.3.2. Flat Parlog Modeli 21
2.7.3.3. Flat Concurrent Prolog (FCP) Modeli 22
2.7.3.4. FCP'nin bir transputer uygulaması 24
2.8. Berk Prolog: Genel Tanıtım 29
2.8.1. Özellikler 30

3. BERK PROLOG: AYRINTILI İNCELEME 41
3.1. Berk Prolog'un Söz Dizimi 41
3.2. İndirgeme (Reduction) 46

4. BERK SİSTEM 50
4.1. Geliştirme Ortamı 51
4.2. Paralel Mimari Sınıfı 51
4.3. Sistemin Topolojisi 52
4.4. Berk Sistem'in Olgu Yapısı 55
4.5. Diğer Yardımcı Olgular 62
4.6. Değişken Numaralandırma Yöntemi 64

4.7. Dondurma (Freezing) 67
4.8. Sunucudan Değişken Talep Edilmesi 72
4.9. Sunucudan Gelen Değişkenin Değerlendirilmesi 75
4.10. Sunucuya Değişken Gönderilmesi 77
4.11. Sunucuya Herhangi Bir Mesajın Gönderilmesi 80
4.12. Gelen Değişkenlerin Sunucu Tarafında Değerlendirilmesi 82
4.13. Bütün Çözüm Ağının Dondurulması 83
4.14. Donan Çözüm Ağının Uyandırılması 85
4.15. Çözüm Ağının Temizlenmesi 85
4.16. Çözüm Ağında Düşme 86
4.17. Çözüm Ağının Tamamen Kapatılması 86
4.18. "Top Level" Durumuna Düşmek 87
4.19. Sunucunun İstemcilerden İleti Toplama Yöntemi 87
4.20. İletilerin İşlenme Tekniği 89
4.20.1."mark" iletisi 90
4.20.2."nop" iletisi 91
4.20.3."shell/1" iletisi 92
4.20.4."debug" iletisi 92
4.21. İndirgeme İşlemi 93
4.21.1.Ön Hazırlık 93
4.21.2.İndirgeme 95
4.21.3.Gövdenin İşlenmesi 98
4.21.4.Yardımcı Bir "berk_fact" Olgusu: dummy 101
4.21.5."fail" fonksiyonu 101
4.21.6."true" Fonksiyonu 102
4.21.7.Klasik Prolog ile Etkileşim 102
4.21.8."unify" Fonksiyonu 103
4.21.9."enque" Kavramı 103
4.21.10. Kullanıcı Girişleri 105
4.21.11. "if/2" Uygulaması 107
4.21.12. Akışların Yazılması (Stream Output) 108
4.22. CPU Tüketimi Analizi 110

5. KÜTÜPHANE YÜKLEMLERİ 112
5.1. Giriş Akışları 112
5.2. Çıkış Akışları 115
5.3. Akışların Birleştirilmesi (Stream Merging) 116
5.4. Keyfi Elemanlı Stream Üretimi 120

6. ÖRNEK UYGULAMALAR 121
6.1. Sınırlandırılmış Akış (Bounded Buffer Streams) 121
6.2. Bir Çakışma Örneği 123
6.3. Klasik Multiples 124
6.4. Basit Kullanıcı Giriş/Çıkışı 126
6.5. Tek Değerin Giriş/Çıkışı 126
6.6. Kullanıcı giriş/çıkışı 127

6.7. Değişken Taşıma 127
6.8. Çakışma 128
6.9. Birleştirme 128
6.10. "fail" durumu 128
6.11. fail/0 129
6.12. if/2 129
6.13. if/2 129
6.14. freeze 130
6.15. Unbounded Buffer 130

7. İLERİKİ ÇALIŞMALAR **131**

8. SONUÇ **132**

KAYNAKLAR **133**

ÖZGEÇMİŞ **138**

KISALTMALAR

CCND	: Committed Choice Non Deterministic
CP	: Concurrent Prolog
FCP	: Flat Concurrent Prolog
GHC	: Guarded Horn Clause
KL	: Kernel Language
GC	: Garbage Collection
PSI	: Personal Sequential Inference Machine
RU	: Reduction Unit
HU	: Host Unit
IPC	: Inter Process Communication
GNU	: Gnu is Not Unix
TCP	: Transmission Control Protocol
UDP	: User Datagram Protocol
IP	: Internet Protocol
CLP	: Constraint Logic Programming
DCG	: Definite Clause Grammer
PVM	: Parallel Virtual Machine
MPI	: Message Passing Interface
PE	: Process Engine
G/Ç	: Giriş/Çıkış

TABLO LİSTESİ

Sayfa No

Tablo 3.1. Berk Prolog'un söz dizimi 44
Tablo 4.1. Önişlemciden geçmiş bir program örneği 59
Tablo 4.2. Dondurma işlemi 70

ŞEKİL LİSTESİ

Sayfa No

Şekil 2.1 : Bağımlı değişkenler ile iki yönlü iletişim 8
Şekil 2.2 : FCP sisteminin soyut yapısı 23
Şekil 4.1 : Berk Sistem'in topolojik yapısı 52
Şekil 4.2 : İnternet ortamında adlandırma 54
Şekil 4.3 : Localhost ortamında adlandırma 54
Şekil 4.4 : Sunucu tarafında değişkenlerin saklanması 63
Şekil 4.5 : İstemci tarafında değişkenlerin saklanması 65
Şekil 4.6 : İstemci tarafında değişkenlerin saklanması 76
Şekil 4.7 : X değişkenine p(Y, Z) teriminin atanması 77
Şekil 5.1 : Keyfi elemanlara sahip bir akışın üretilmesi 119

AN EXTENSION OF PROLOG TO PARALLEL LOGIC PROGRAMMING LANGUAGE

SUMMARY

Parallel logic programming emerged as a research field following the success of Prolog as the prominent logic programming language. The Japanese Fifth Generation Computer Project initiated in the early 1980s, in which parallel logic programming language was selected as the implementation tool, accelarated the reseach in this field. During the following 10 years important advances took place in "stream and-parallel" languages. After the mid 1990s, the research in parallel logic programming diverted to "implicit parallelism".

In our work we introduce a new parallel logic programming language as a new member of CCND programming languages. We named our parallel logic programming language as "Berk Prolog" and its MIMD implementation as the "Berk System".

Berk Prolog differs from other parallel logic programming langauges developed until today in many respects. It differs from conventional parallel logic programming languages primarily in its syntax and semantics.

Some of the main differences can be summarized as follows: Berk Prolog differs from other related languages in its synchronization mechanism, in wich a process cannot be suspended by head unification or guard evaluation. A process can be invoked if all input variables are bound. Guard evaluation is the main subject of computation rather than clause selection.

Berk Prolog introduces the concept of conditional demand of external variables. It also brings flexibility on clause mapping and distribution of processes.

By its guard evaluation, the sequential Prolog is treated as a subset of Berk Prolog. As Berk Prolog is a stable system, if-then-else and related predicates can be implemented quite easily.

The Berk System which we have developed as an implementation of Berk Prolog, is a model that works on TCP/IP network medium. In this model we introduce the concepts of solution network and network heap. Some familar benchmark programs have been written in Berk Prolog and executed in Berk System successfuly.

PROLOG'UN PARALEL MANTIK PROGRAMLAMAYA GENİŞLETİLMESİ

ÖZET

Mantık programlamanın, Prolog programlama dili ile vücut bulması ile birlikte, paralel mantık programlama çalışmaları da başlamıştır. 1980'lerin başında Japonların beşinci kuşak bilgisayar projesinde geliştirme dili olarak paralel mantık programlamayı seçmesi, bu konudaki çalışmalara çok büyük bir ivme kazandırmıştır. Takip eden 10 yıl içinde akışlı ve-paralel dillerde büyük gelişemeler yaşanmıştır. 1990'ların ortalarından sonra paralel mantık programlama dillerinde araştırmalar örtük paralelliğe doğru kaymıştır.

Bu çalışmamızda, CCND dil ailesi içinde bulunan, yeni bir paralel mantık programlama dili öneriyoruz. Bu paralel mantık programlama dilini Berk Prolog ve MIMD mimarideki uygulamasını da Berk Sistem olarak adlandırdık.

Berk Prolog bugüne kadar tanıtılmış paralel mantık programlama dillerinden pek çok yönden farklılıklar göstermektedir. Öncelikle Berk Prolog, döz-dizimi ve semantik olarak klasik dillerden ayrılmaktadır.

Bazı temel farklılıkları şöyle özetleyebiliriz: Senkranizasyon mekanizması farklıdır. Bir proses baş veya guard tarafından askıya alınamaz. Bir proses ancak bütün giriş değişkenleri belli olduktan sonra çağrılabilir. Bundan dolayı proses çağrısı hiç bir zaman prosesleri askıya almaz. Guard kısmı, cümle seçiminden çok esas hesaplama kısmını oluşturur.

Şartlı olarak değişken talep etme ve körleme bekleme kavramları ortaya atılmıştır. Kullanıcılara, cümle ve proses dağıtımı konusunda çok büyük esneklikler getirilmiştir.

Guard çağrıları sayesinde sıralı Prolog, paralel Prolog'un bir alt kümesi haline getirilmiştir. Kararlı bir sistem olduğu için, if-then-else yapıları ve bu yapılarla ilgili diğer temel fonksiyonlar kolaylıkla gerçeklenebilmiştir.

Berk Prolog'un bir yürütme modeli olarak Berk Sistemi geliştirdik. Berk Sistem TCP/IP ortamında çalışan dağıtık ve çok farklı bir uygulama modelidir. Bu modelde çözüm ağı ve ağ alanı gibi kavramlar sunulmuştur. Paralel mantık programlamada kullanılan bazı kıyas programları Berk Prolog'da yazılmış ve Berk Sistem üzerinde başarı ile yürütülmüştür.

1. GİRİŞ

Mantık programlamanın başlangıcı, 1967 yılında A. Robinson'un mantıkta yeni bir metot olan "çözülüm teorem ispatlama" metodunu geliştirmesine kadar geriye götürülebilir. 1970'lerin başlarında Alain Colmaurer ve Robert Kowalski'nin ilk mantık programlama dili olan Prolog'un kavramlarını tanıtmasından sonra bu alandaki gelişmeler hız kazanmıştır. 1975'te David Warren ilk Prolog derleyicisinin bilgisayar uygulamasını gerçekleştirmiştir. Bundan beş yıl kadar sonra da Avrupa ve Japonya'da paralel Prolog derleyicileri üzerine çalışmalar başlamıştır.

Japonların Beşinci Kuşak Bilgisayar Projesi sayesinde paralel Prolog en kuvvetli, en görkemli günlerini 1982-1992 yılları arasında yaşamıştır. Beşinci Kuşak Bilgisayar Projesinde paralel işleme ile zeki bir sistem geliştirmek için paralel mantık programlama bir araç olarak seçilmiştir.

Başlangıçta, Prolog'u paralelleştirme çalışmaları, Japonların Beşinci Kuşak Bilgisayar Projesi doslayısıyla açık parallellik doğrultusunda idi. Fakat açık paralel sistemler hem sentaks hem de semantik itibari ile Prolog programlamadan son derece uzaktı. Japonların projelerini tamamlamasından sonra [17], açık paralel sistemler konusunda yapılan çalışamalar nerede ise son bulmuştur.

Prolog programlamaya daha yakın olan örtük paralel sistemler üzerinde çalışmalar da daha sonraları başlamıştır. Bu çalışmalardan MUSE [14] ve Aurora [15] bazı özel koşullar altında sıralı Prolog'un bütün özelliklerini sağlamaktadırlar.

En klasik açık paralel sistemler Parlog [23] [27], Concurrent Prolog [20] ve GHC [24] [25] [26] sistemleridir. Bugün Parlog hariç diğer sistemler kullanılmamaktadır. Parlog sistemi ise üniversitelerin bilgisayar labratuarlarından dışarıya çıkamamış, yaygınlaşamamıştır.

Bu tez çalışması, 1990'lardan sonra artık nerede ise tamamen terkedilmiş olan açık paralel mantık programalama sistemlerine yeni bir bakış açısı getirmek amacı ile yapılmıştır.

Bunun için öncelikle, açık paralel bir sistem olan Berk Prolog'un sentaksı verilmiştir. Çok geniş bir anlamda semantik tanıtılmıştır.

Diğer açık sistemlerden farklı olarak Berk Prolog kısıtlandığında standard Prolog'a erişilmektedir. Bundan dolayı Berk Prolog'a standard Prolog'un genişletilmesi gözü ile bakıyoruz.

Bu tez çalışması ile, ayrıca Berk Prolog'un TCP/IP ağı altında çalışan bir uygulaması da gerçekleştirilmiştir. Bu uygulamayı Berk Sistem olarak adlandırıyoruz.

Berk Prolog'un önerilmesi ve Berk Sistem'in yapılmasındaki temel amaç, standard Prolog derleyicilerinin üzerinde hiç bir değişiklik yapmadan, bu derleyicileri paralel hale getirebilmektir.

Bu tezin ikinci bölümünde kısaca sıralı ve paralel Prolog sistemleri üzerinde durulacak ve bu konuda yapılan çalışmalar özetlenecektir. Daha sonra açık paralel sistemlerin ortak özelliklerinden ve prensiplerinden bahsedilecek ve klasik açık paralel diller ve uygulamaları ayrıntılı olarak incelenecek ve Berk Prolog'a giriş yapılacaktır.

Üçüncü bölümde Berk Prolog dördüncü bölümde ise Berk Sistem ayrıntılı olarak incelenecek, devam eden bölümlerde ise kütüphane yüklemleri ve örnek programlar verilecektir.

2. PARALEL MANTIK PROGRAMLAMA

Bu bölümde sıralı Prolog ve paralel Prolog'a kapalı geçişten bahsedilecektir. Ayrıca klasik paralel mantık programlama dilleri ve uygulamaları üzerinde durulacak ve bu tezin esasını teşkil eden Berk Prolog sistemine giriş yapılacaktır.

2.1. Sıralı Prolog

Prolog[1] programlama, mantık programlama dilleri (logic programming languages) içinde bilinen en klasik bilgisayar programlama dildir. Prolog programlama dili ilişkisel bir programlama dili (relational language) olup son derece basit üç mekanizma üzerine üzerine inşa edilmiştir: Birleştirme (unification), otomatik arama ve otomatik geriye-iz-sürme (backtracking).

İlk Prolog uygulaması 1967 yılında Aberdeen Üniversitesinde gerçekleştirilmiştir [1]. Bir arı Prolog (pure Prolog) uygulaması olan bu çalışma yaygınlaşmamıştır. Alain Colmaurer ve Robert Kowalski'nin 1970'lerin başında Prolog programlamanın kavramlarını tanıtmasından sonra gelişmeler hız kazanmıştır. Colmaurer ve grubu tarafından pek çok Prolog sistemi gerçekleştirilmiştir. İlk sistem 1972 yılında ALGOL programlama dili ile bir yorumlayıcı (interpreter) şeklinde yazılmıştır. İkinci sistem ise 1973 yılında FORTRAN ile G. Battani tarafından yapılmıştır. 1983 yılında David Warren'in yayınlamış olduğu teknik bir rapor [2] ile Prolog programlamanın yeni bir çağı başlamıştır. Warren'in bu teknik raporundan sonraki 10 yıl içinde Prolog'daki gelişmeler, Peter Van Roy'un [3]'deki makalesinde pek çok detayı ile bulunabilir. Ayrıca Prolog ve mantık programlama konusunda [5], [7], [8], [9], [10], ve [11] numaralı referanslarda verilen kitaplara başvurulabilir.

[1]Prolog, PROgramming in LOGic kelimelerinden türetilmiştir.

2.2. Warren'in Soyut Makinesi

David Warren,[2] [2]'deki teknik raporunda, Prolog için bir yürütme modeli tanıtmıştır. Bu tanıtılan modele *Warren'in Soyut Makinesi* (The Warren Abstract Machine) veya kısaca *WAM* denilmektedir. WAM yürütme modeli günümüzdeki hemen hemen bütün Prolog derleyicileri için bir standard haline gelmiştir. WAM yürütme modeli [4], [5], [6], call/return gibi kontrol komutlarından get/put gibi birleştirme komutlarından ve try/retry gibi geriye-iz-sürme komutlarından oluşmuştur. Ayrıca bu yürütme modeli üzerinde pek çok iyileştirmeler (optimizations) tanıtılmıştır. WAM yürütme modeli bu tezin kapsamı dışında olduğu için üzerinde durulmayacaktır. Sadece konu bütünlüğünü sağlamak için burada değinilmiştir.

2.3. Paralel Prolog

Prolog'u paralelleştirme çalışmaları [13], Prolog'un ortaya çıkışı ile başlamıştır. Nasıl ki WAM modelinin keşfi ile 1983-1993 yılları arasında Prolog[3] altın yıllarını yaşamışsa, Japonların Beşinci Kuşak Bilgisayar Projesi sayesinde paralel Prolog en kuvvetli, en görkemli günlerini 1982-1992 yılları arasında yaşamıştır. Beşinci Kuşak Bilgisayar Projesinde paralel işleme ile zeki bir sistem geliştirmek için paralel mantık programlama bir araç olarak seçilmiştir. İşin ilginç tarafı WAM modeli ile Japon'ların paralel Prolog yürütme modelleri arasında hiç bir ilintinin olmamasıdır. Bu projede klasik mantık programalama dili olan Prolog yerine bir CCND dili seçilmiştir. Bu konuya ileriki bölümlerde tekrar değinilecektir.

[2]Çok ilginçtir ki, Prolog konusunda ünlü bir başka bilim adamının da adı David S. Warren'dir. [5]

[3]Bu tez boyunca geçen Prolog kelimesi, sıralı Prolog'a (sequential Prolog) işaret etmektedir.

Prolog, bir mantık programlama[4] dilidir. Mantık programlama dillerinin en büyük özelliklerinden biri bunların tek atamalık (single value assignment) diller olmasıdır. Prolog programında bulunan bir değişken, bir çözüm bulunana kadar yalnız ve yalnız bir kez değer alabilir. Diğer bir deyişle Prolog programlamada yıkıcı atamalara (destructive assignment) izin verilmez[5]. Değer almış bir değişkene yeni bir değer atanmaya çalışıldığında atama değil birleştirme (unification) işlemi yapılır. Ayrıca değişken sadece bir kez değer aldığı için, alternatif çözüm denemelerinde, değişkene verilen değer geri alınır (trail) ve değişken ilk haline döner.

Ayrıca mantık programlamada icranın sırasını değiştirdiğimiz zaman elde edilecek sonuç değişmez. Bir problemin çözümünde p, q çağrısı(atomic goal) ile q, p çağrısı aynı sonucu verir. Hem tek değerli atama hem de çözüm sonucunun icra sırasından bağımsız olması bize çeşitli yürütme teknikleri geliştirmemize imkan sağlar. Yapısal programlama dillerinde böyle bir yaklaşım mümkün değildir.

Farklı yürütme tekniklerinden birisi de paralel yürütmedir. Çünkü Prolog'un yukarıda saydığımız özelliklerinden dolayı paralel veya sıralı yürütülmesinden dolayı elde edilecek sonuçlar değişmeyecektir. Böylece klasik bir Prolog programını hiç bir değişikliğe uğratmadan paralel olarak yürütebiliriz.

2.4. Örtük Paralellik

Bir Prolog programını hiç değiştirmeden, diğer bir deyişle kullanıcının hiç bir müdahalesi olmadan paralel olarak yürütmeye *örtük paralel yürütme* (implicit exploitation of parallelism) denir. Bu yürütme modelinde bütün yük derleyicilerin üzerindedir. Gerekli paralelleştirme işlemini, derleme veya yürütme zamanında, derleyiciler yapar, kullanıcı paralel yürütme için herhangi bir gayret sarfetmez.

[4] *Mantıksal* demek doğru değildir. Çünkü, orjinal terim *logical programming* değil *logic programming*'tir.

[5] Flag, global değişken veya yan-etkiler (side effects) [12] gibi özel durumlar hariç.

Örtük paralellik dört ayrı gruba ayrılır.

Birleştirme paralelliği (unification parallelism)

İki terim birleştirileceği zaman, argümanları paralel olarak birleştirilir.

Örneğin p(P1, P2, P3) terimi ile p(Q1, Q2, Q3) terimi birleştirilecek olsun. P1 ile Q1, P2 ile Q2 ve P3 ile Q3 paralel olarak birleştirilir.

Fakat bu yöntem çok ince taneli (very fine grained) bir paralelleştirme olduğu için özel donanımlara muhtaçtır. Bundan dolayı birleştirme paralelliği fazla bir ilgi görmemiştir.

Veya paralelliği (Or paralellism)

Sadece alternatif çözümleri olan problemlerde ortaya çıkar. Bir yüklem (predicate) birden fazla cümleden (clause) oluşsun. Bir çağrı, aynı anda, paralel olarak birden fazla cümlenin baş kısmı (head of clause) ile birleştirilir. Eğer başarılı olursa her bir proses yürütmeye devam eder.

Bağımsız-Ve paralelliği (Independent and parallelism)

Yürütme zamanı sırasında, *Ve* operatörü (and operator) ile bağlı çağrılarda bulunan değişkenlerin kesişimi boş küme ise bu paralellik ortaya çıkar. Örneğin *p(X), q(Y), r(Z)* çağrılarında, hiç bir değişken bir diğer çağrıda gözükmez. Çağrılar, değişken bazında birbirlerinden bağımsızdırlar. Çağrılar arasında veri bağımlılığı mevcut değildir.

Bağımlı-Ve paralelliği (Dependent and parallelism)

Yürütme zamanı sırasında, *Ve* operatörü ile bağlı, en az iki çağrıda en az 1 adet ortak değişken mevcut ise bu tip paralellik meydana çıkar. Bu iki çağrı arasında veri bağımlılığı (data dependency) mevcuttur. Bu paralellik iki türlü icra edilir.

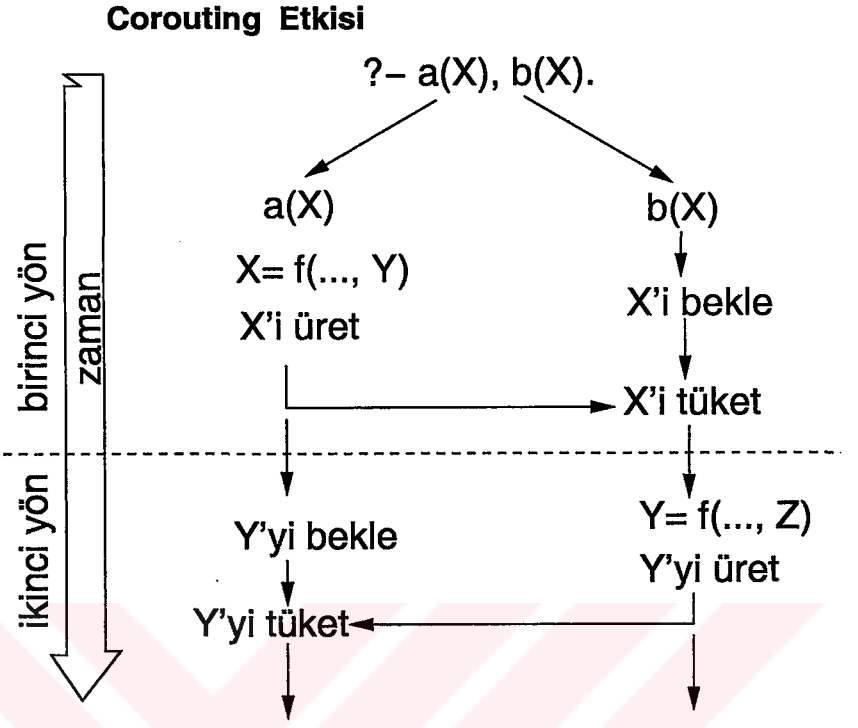
Birinci durumda ortak değişkene sahip her iki çağrı da bağımsız olarak, paralel yürütülür. Paralel yürütmede ortak değişkene ihtiyaç duyulduğunda, her iki

tarafın üretmiş olduğu değer (the value of the variable) birleştirilir. Bu tekniğe "sonradan birleştirme" (back unification) denir.

İkinci durumda ise, yürütme sırasında ortak değişkene erişildiğinde proseslerden biri değişkeni üretir. Diğer proses de bu değişkeni kullanır. Bu tekniğe üretici/tüketici (producer/consumer) yöntemi denir. Bu paralellikte çağrılar arasında her zaman veri bağımlılığı mevcuttur. Genelde, çağrılardan sadece ve sadece biri üretici, diğer bütün çağrılar ise tüketici konumundadır. Çağrılar arasındaki bu bağımlı değişken (dependent variable) bir tür tek yönlü iletişim kanalı (communication stream) haline gelir. Bundan dolayı bağımlı-ve paralelliğe, akışlı paralellik (stream paralellism) de denir. Ayrıca iki çağrı arasında bağımlı değişkenler vasıtası ile iki yönlü iletişim kurmak da mümkündür. Şekil 2.1'de iki yönlü paralellik temsil edilmiştir.

Şekil 2.1'de gösterilen akış için ?- a(X), b(X). şeklinde bir sorgu (query) verilmiş olsun. Görüleceği üzere, a/1 ve b/1 çağrıları, ortak X değişkeni yüzünden, bağımlı-ve paralelliğe sahiptirler. a/1 ve b/1 çağrıları farklı iki makine üzerinde işlenecek olsunlar. Önce a/1'de bulunan X değişkenine f(Y) atanmış olsun. Y değişkeninin, şimdilik içeriksiz (unbounded) olduğunu varsayalım. Bu sırada b/1'in bulunduğu ikinci makinede Y değişkeni beklenmektedir. a/1'in işlendiği ilk makinede elde edilen X= f(Y) değeri ikinci makineye gönderilir. İkinci makine f(Y) değerini alır ve gerekli bir yerde kullanır, diğer bir deyişle tüketir. Fakat bu sırada birinci makine Y değerini beklemektedir. İkinci makine Y değerini üretir ve birinci makineye gönderir. Fakat üretilen Y değeri içine içeriksiz bir Z değişkeni ekler. Bu işlem böylece sürer gider. Böylece iki proses arasında veri akışı sağlanmış olur. Bu özelliğe *corouting effect* denir.

Bağımlı-ve paralelliği sağlamak için mutlaka corouting etkisini kullanmaya gerek yoktur. Diğer uygulama bağımsız-ve gibi yapılabilir. Corouting yöntemi bağımlı-ve paralel uygulamalarda çok etkilidir. Fakat bu yöntemde *non-determinism* uygulaması çok büyük bir problemdir. Bu problemi çözmek için CCND dillerinde, *non-determinism* hemen hemen iptal edilmiştir. Bu uygulama şekline *don't care*



Şekil 2.1: Bağımlı değişkenler ile iki yönlü iletişim

nondeterminism denir. Bu metotta deterministik olmayan bir seçimle, yakalanan ilk cümle dikkate alınmaktadır.

Üretici/tüketici ilişkisine sahip, bağımlı-ve paralelliğine *akış paralelliği* (stream paralellism) de denir. Akış paralellliği, CCND dillerinin de temelini teşkil eder.

Paralel mantık programlamanın amacı mümkün olduğu kadar çok paralellik ile en iyi performansı elde etmektir. Yukarıda bahsi geçen dört paralellik biçimi de birbirlerinden tamamen bağımsızdırlar. Bir paralel Prolog programı aynı anda tamamını veya en az birini uygulayabilir. Çünkü biri diğerini etkilemez. Bunların hepsini birleştiren etkili bir sistemin gerçekleştirilmesi pratik olarak çok zordur.

Özellikle 1990'dan sonraki yıllarda paralel Prolog konusundaki çalışmalar örtük paralelliğe doğru kaymıştır. Sadece örtük paralellik ve bu konuda yapılmış olan çalışmaları inceleyen geniş çaplı bir araştırma makalesi [16]'da bulunabilir. Fakat örtük paralellik bu tezin kapsamı dışında tutulmuştur.

2.5. Bağımlı-Ve Paralelliğin Uygulanmasındaki En Genel Problem

Şekil 2.1'e tekrar geri dönelim. b/1 çağrısını yürüten ikinci işlemci (processor), X değişkenini beklemektedir. Fakat X'i beklemesi gerektiğini nereden bilecektir? Belki de kendisi X değişkenini üretecek ve diğer işlemci üretilen bu değişkeni bekleyecektir. İşte bağımlı-ve paralellikte ortaya çıkan ana problemden birisi, değişkeni kimin üreteceği veya diğer bir deyişle, ilgili değişkenin üretici mi yoksa tüketici mi olduğunun tespitidir.

Bir an için bu problemin çözüldüğünü ve birinci prosesin X değişkenini atadığını varsayalım. Bu durumda birinci proses X için üretici, ikinci proses ise X için tüketici durumunda olacaktır. Şimdi birinci prosesin üretmiş olduğu X değişkenini ikinici prosese gönderdiğini kabul edelim. Bu durumda ikinci prosesin ve, eğer varsa bu değişkeni bekleyen diğer bütün tüketici proseslerin ayağa kalkıp kaldıkları yerden yürütmeye devam etmeleri gerekmektedir. İşte ikinci ana problem de bir değişkenin atanmasını bekleyen proseslerin etkin bir biçimde uyandırılmasıdır.

2.6. Paralel Mantık Programlamada Yaklaşımlar

Bazı paralel Prolog sistemleri sıralı Prolog'da olduğu gibi çalışırlar: Elde edilen nihai sonucun yazılması ve yürütmenin soldan sağa, yukarından aşağı icra edilmesi gibi. Böylece, üzerinde hiç bir değişiklik yapılmadan, bir Prolog programı paralel olarak yürütülebilmektedir. MUSE [14], Aurora [15] gibi paralel Prolog sistemleri özel koşullar altında sıralı Prolog özelliklerini sağlarlar. MUSE sistemi ilk ticari paralel Prolog sistemidir.

Japonların beşinci kuşak bilgisayar projesinin[6] tamamlanması [17] ile CCND dilleri konusunda yapılan çalışmalar da neredeyse son bulmuştur. Çalışmalar

[6] 1992 yılında düzenlenen son bir konferans ile çok ünlü beşinci kuşak bilgisayar projesine son verilmiştir. Bu konferansa 2000 civarında bilim adamı katılmıştır.

örtük paralel sistemler üzerinde yoğunlaşmıştır. Bu tür sistemlerin uygulamaları ve problemleri [16]'da bulunabilir.

CCND dilleri açık paralelliğe (explicit parallelism) sahiptir. Açık paralel dillerde kullanıcılar üzerine bazı yükler binmektedir. Paralelliği sağlayabilmek için kullanıcı, yazmış olduğu paralel bilgisayar programına[7] bazı eklentiler yapmak zorundadır. İşte kullanıcının dışarından, açıkça bir müdahalesi mevcut olduğu için bu tür paralelliğe açık paralellik denir.

Örtük paralellik kullanıcı açısından bakıldığında çok tercih edilen bir tekniktir. Çünkü kullanıcı yazdığı Prolog programı üzerinde hiç bir değişiklik yapmayacaktır. Bütün yük paralel derleyiciyi üzerindedir.

Açık paralellikte ise yükün büyük kısmı kullanıcı üzerindedir. Kullanıcı en azından hangi değişkenin üretici hangilerinin tüketici olduğunu yüklem bazında veya cümle bazında seçmek zorundadır. Ayrıca örtük paralel sistemler, kapalı paralel sistemlere nazaran, standrad Prolog'dan daha da uzaklaşmaktadırlar.

2.7. CCND Dilleri

CCND dilleri, akışlı ve-paralel (stream and-parallel computational model) bir hesaplama modeli sunarlar. Daha önce de kısaca bahsedildiği gibi, bu modelde, çağrılar arasında bulunan ortak değişkenler prosesler arasında bir iletişim kanalı (communication channel) kurarlar. Bir çağrının işlenebilmesi için bütün giriş değişkenlerinin belirli, atanmış olması gereklidir. Bu tanım, açıkça CNND dillerini veri-akışlı diller (data-flow languages) içine katmaktadır.

[7]Artık bu bilgisayar programına, Prolog programı diyemiyoruz. Çünkü CCND dilleri ile yapılan paralel Prolog sistemlerinin aslında Prolog ile pek de bir alakası mevcut değildir. Çünkü iki yönlü olması gereken birleştirme işlemi tek yönlüdür. CCND dilleri genelde kararlı (stable) değildir, yani yüklemlerin seçiliş sırası her zaman yazım sırasında değildir. Ayrıca bactracking kavramı tamamen iptal edilmiştir. *Non-determinism* ise *don't care non-determisinism* kavramı ile gayet uyumlu bir hale getirilmiştir.

Ayrıca bu dillerin bir diğer özelliği ise daimi (perpetual) proseslere destek vermesidir. Aşağıda, akışlı ve-paralel ve daimi proseslere sahip bir örnek[8] verilmiştir.

```
natural(X, [Xs | Y]) :-  Xs is X,
                         X0 is X+1,
                         natural(X0, Y).

double([X  | Y], Xd) :-  Xd is X*2,
                         double(Y, Xd).

?- double(X, Y), natural(0, X).
```

Yukarıda *natural/2* yüklemi bir doğal sayı üreticisidir. Üretilen doğal sayılar ikinci argümanda bulunan listenin sonuna eklenir. *natural(0, X)* sorgusu, sıfırdan başlayarak sonsuza kadar doğal sayı üretir.

Örnekteki *double/2* yüklemi ise, natural/2 tarafından üretilen doğal sayıları *tüketir*. Aynı zamanda üretilen her doğal sayıyı 2 ile çarparak başka bir üretim yapmaktadır.

Yukarıdaki her bir yüklem sıralı bir Prolog makinesinde problemsiz olarak, tek tek çalıştırılabilir. Fakat istenilen amacın elde edilebilmesi için, yani doğal sayı dizisi ve bunun iki katının üretilebilmesi için mutlaka paralel çalıştırılması gerekir.

Farzedelim ki örnekteki, *?- double(X, Y), natural(0, X).* sorgusu paralel bir makinede aynı anda çalışmaya başlasın. Bu durumda X değişkeni iki proses arasında bir iletişim kanalı kuracak ve bir tarafta üretilen doğal sayılar, X üzerinde diğer prosese geçecek ve orada tüketilecektir. Yukarıda verilen örnekte, üretim işi, sistem kaynakları tükenene kadar devam edecektir.

[8]Kaynak [17], sayfa 240

Yukarıdaki örnekte, natural/2 ve double/2 yüklemleri sürekli olarak kendilerini çağırmakta, diğer bir deyişle devamlı olarak yürütme modeli içinde kalmaktadırlar. Bu süreklilikten dolayı bu tür proseslere *daimi prosesler* denir.

CCND dilleri üretici/tüketici tipindeki problemler için en uygun dillerdir. [20], [21]. Bu tür problemlerin en çok meydana çıktığı alanlar ise bilgisayar işletim sistemleridir. Japonların Beşinci Kuşak Bilgisayar Projesi'nde [19], CCND dillerinin işletim sistemi olarak kullanımı başarı ile gerçekleştirilmiştir.

Yukarıdaki örnekte, bir an için natural/2 ve double/2 yüklemlerinin sıralı Prolog'da ve Unix işletim sistemi altında, bağımsız olarak derlendiğini varsayalım. Bu durumda *?- double(X, Y), natural(0, X).* sorgusu yerine, temsili olarak *$ natural | double* yazılabilir. Bu durumda unix işletim sisteminde iki bağımsız proses açılacak ve birinde elde edilen yani üretilen değerler diğerinde tüketilecektir. Unix işletim sistemindeki *pipe* özelliği gibi pek çok özellik kolaylıkla CCND dilleri tarafından gerçeklenebilir [20], [21]. CCND dilleri ile ilgili çok geniş bir çalışma [22]'de bulunabilir.

2.7.1. Ortak Özellikler

Bütün CCND dillerinin söz-dizim kuralı (syntax) 2.1'deki yazım şekline uyar.[9]

$$H \ :- \ G_1, G_2, ..., G_n \mid B_1, B_2, ..., B_m \ . \quad n, m \geq 0 \tag{2.1}$$

H, G_i ve B_i ifadeleri klasik Prolog'un terimleri gibidir. Tek bir atomdan veya bir adet yüklem adı (functor) ve belirli sayıda argümandan (arity) oluşur.

2.1 ifadesi üç farklı şekilde meydana çıkar. 2.1 ifadesinde eğer en az 2 adet gövde çağrısı mevcutsa, $m \geq 2$ ise, bu ifade *genel cümle* (general clause) adını alır. 2.1 ifadesi, tam bir adet gövde çağrısına sahipse *yinelemeli* (iterative), hiçbir

[9]Dec-10 Prolog kurallarına göre verilmiştir. H, G ve B harfleri, Head, Guard ve Body kelimelerinin baş harfleridir. "|" işaretine seçme operatörü (commit operator) denir.

gövde çağrısına sahip değilse *birim cümle (unit clause)* adını alır. Hiç bir gövde çağrısı olmayan cümlede, gövdeyi *true* çağrısı ile temsil ediyoruz. Çünkü seçme operatöründen sonra en az 1 adet eleman bulunmalıdır. Bu paragrafta yazılan ifadeler aşağıda özetlenmiştir.

$$Genel \quad H \ :- \ G_1, G_2, ..., G_n \mid B_1, B2, ..., B_m \ . \quad n \geq 0, m \geq 2 \tag{2.2}$$

$$Yinelemeli \quad H \ :- \ G_1, G_2, ..., G_n \mid B_1 \ . \quad n \geq 0 \tag{2.3}$$

$$Birim \quad H \ :- \ G_1, G_2, ..., G_n \mid true \ . \quad n \geq 0 \tag{2.4}$$

Ayrıca bu dillerde *guard* özelliği ve veri-akış senkronizasyonu (data-flow synchranisation) prensibi her zaman ortaktır.

Cümlelerde bulunan *guard* çok önemli bir özelliğe sahiptir. Genelde bu tür diller, *guard*'ın uygulanma tekniğine göre birbirlerinden ayrılırlar.

Guard'ın cümle içindeki görevi kısaca şöyle açıklanabilir. Bir çağrı yapıldığında önce çağrıdaki terim ile H birleştirilir (head unification). Bu işlem başarılı olursa seçme operatörüne (commit operator) kadar olan bütün çağrılar sıralı veya paralel bir biçimde yürütülür (guard evaluation). Eğer bu yürütmenin tamamı başarılı olursa seçme operatörü devreye girer. Aslında bu operatör bir iş yapmaz. Bu operatörü sıralı Prolog'daki kesme (cut) operatörüne benzetebiliriz. Bu adımda sanki bir kesme işlemi yapılıyormuş gibi bütün alternatif çözüm yolları (choice points) iptal edilir. Artık bu adımdan sonra asla geri dönüş yoktur, geriye-iz-sürme yoktur. İşte *non-determinism*'in bu tip uygulamasına *don't care nondeterminism* denir.

Örtülü paralel dillerde genelde geriye-iz-sürme yapılabilir. Fakat CCND dillerinde geriye-iz-sürme işlemi sadece ve sadece seçme operatörüne kadar geçerlidir. Bu adımdan sonra geri dönüş yoktur. Bundan dolayı CCND dillerinde seçim operatöründen sonra herhangi bir çağrı başarısız olursa (failure of a goal) bütün

hesaplama başarısız (whole computation failure) olacaktır. Çünkü alternatif çözüm yolları, *guard* hesaplama bittikten hemen sonra iptal edilmiştir.

Seçme operatörüne kadar yapılan işlemler bir CCND dilinin karakteristiğini tespit eder. Sonraki bölümlerde *guard* işleme üzerinde ayrıntılı olarak durulacaktır.

Seçme operatörüne kadar yapılacak işlemler her ne kadar uygulamadan uygulamaya değişse de, bu operatörden sonra yapılan işler bütün dillerde aynıdır. Bu opertörden sonraki her bir çağrı paralel olarak işletilir.

Yukarıda söz-dizimi verilen cümle şu anlamı (declarative semantics) ifade eder.

Cümle içindeki değişkenlerin bütün değerleri (value of the variable) için, Eğer $G_1, G_2, \ldots G_n$ *ve* $B_1, B_2, \ldots, B_m$ *doğru ise* H *doğrudur.*

Gaurd çağrılarına belli kısıtlamalar getirilebilir. Sadece daha önceden tanıtılmış yüklemlere izin verilir. Bu özelliğe sahip dillere *Flat CCND* dilleri denir. Flat CCND dillerinde genelde sadece test yüklemleri mevcuttur. Diğer bir deyişle yüklemler gövde veya baş değişkenlerine bir atama yapmaz. Bir Flat CCND dilinin güvenli (safe) olabilmesi için, *guard* kısmındaki çağrıların cümle başında (head of the sentence) bulunan giriş değişkenlerine (input variables) atama yapmaması (binding variable) gerekmektedir.

2.7.2. Klasik CCND Dilleri

Literatürde üç adet CCND dili mevcuttur. Parlog, Concurrent Prolog ve Guarded Horn Clause (GHC). Aşağıdaki bölümlerde her bir dilin özellikleri kısaca incelenecektir.

2.7.2.1. Parlog

Parlog [23] [27] dilinde, birleştirmenin yönünü tespit etmek amacı ile yüklem seviyesinde mod tanımı yapılır. Yüklemin her bir argümanı giriş veya çıkış modları (? ve ^) ile tanıtılır.

Baş birleştirilmesi (head unification) sırasında giriş argümanıyla karşılaşılırsa ve bu değişkene değer atanmamış ise bu proses hemen askıya alınır. Eğer yürütme seçim operatörüne kadar gelmiş ise gövde çağrıları işlenmeden evvel çıkış argümanlarına atama yapılır. Böylece cümlenin seçimi garanti edildikten sonra çıkış argümanları kullanıma açılır.

Parlog programları güvenli (safe) olmak zorundadırlar. Diğer bir deyişle *guard* çağrıları asla giriş modunda bulunan argümanları atamamalıdır.

Program taşıma tekniği ile Parlog programları kernel Prolog'a çevrilir. Kernel Prolog daha basit bir dildir ve farklı mimarileri desteklemektedir. Ayrıca kernel Prolog'da mod tanımı artık mevcut değildir. Kernel Prolog'a çevirme işleminden sonra *flat* dile çevirme işlemi yapılır. Böylece Parlog modeli iyice basitleştirilmiş olur.

Cümlenin baş kısmı için tek yönlü birleştirme (one way unification) geçerlidir. *Guard* işleme kısmında ise standard birleştirme yapılır. *Guard* işleme sırasında genelde Veya (OR) prosesleri üretilir. Bütün *guard* işlemleri başarılı olursa artık bu cümle geri dönüşü olmayacak bir biçimde seçilmiştir. Daha sonra gövde (body) içindeki her bir çağrı (atomic goal) için bir proses açılır.

Flat Parlog'da giriş değişkenleri *guard* tarafından asla atanmazlar ve çıkış değişkenleri sadece ve sadece cümlenin seçimi garantilendikten sonra atanırlar.

2.7.2.2. Concurrent Prolog

Concurrent Prolog (CP) [20] veri-akışı (data-flow) kısıtı olarak değişken seviyesinde salt-okunur takısını[10] (read-only annotation) kullanır. CP'nin senkronizasyon mekanizmasını oluşturan salt-okunur değişkenler veri-akış senkronizasyonunun (data-flow synchronizaiton) [39] genelleştirilmiş bir halidir.

CP hesaplama modeli (computational model) aşağıdaki özelliklere sahiptir.

- Bir CP sistemi Ve ile bağlı çağrılardan (conjunctive goal) oluşur. Örneğin, ?- p(X), q(X?, Y), r(Y?).
- Proses'leri birim çağrılar (unit goal) oluşturur, p(X) gibi.
- Argüman değerleri bir prosesin durumunu (process state) verir.
- Proses hesaplaması (process computation), indirgeme (reduction) ile yapılır.
- Prosesler arası iletişim (process communication) ortak değişkenler ile yapılır. p(X), q(X?, Y) örneğinde X değişkeni p/1 ve q/2 prosesleri arasında verinin taşınmasını sağlar.
- Prosesler arası senkronizasyon (process synchronization) salt-okunur değişkenler yardımı ile yapılır. r(Y?) çağrısında eğer Y değişkenine herhangi bir atama yapılmamış ise r/1 prosesi askıya alınır. Böylece Y değişkeninin önünde bulunan salt-okuma eki ile Y değişkeninin hazır olarak gelmesi beklenmektedir. Bu soru ekinden dolayı Y değişkenine asla atama yapılamaz.
- Prosesin başarısız olması (process failure), herhangi bir prosesin *fail* mesajı ile son bulmasıdır.
- Bütün proseslerin durumlarının birleşimi bütün sistemin durumunu (system state) oluşturur.

[10]Bu takı "?" ile gösterilir.

CP'de, Parlog gibi yüklem seviyesinde mod tanımı mevcut değildir. Salt-okunur takı, sadece birleştirme sırasında ele alınır. Çünkü bir giriş değişkeni, diğer bir deyişle salt-okunur bir değişken, bir terimin herhangi bir yerinde ortaya çıkabilir. Ayrıca *guard* çağrıları bütün prosesi askıya alabilir. Birleştirme işlemi de zamana bağlıdır, ilk anda başarısız (fail) olan bir iş daha sonra başarılı olabilir. Diğer bir deyişle, başarısızlık salt-okunur değişkenin henüz prosese ulaşmamasından kaynaklanıyor olabilir. Bu durumda ileriki bir zamanda bu prosesin her zaman başarılı olma ihtimali vardır. Fakat diğer durumlardaki başarısızlıklarda, zaman önemli değildir. Proses kesin olarak başarısız bitmiştir.

Değişken atama için hiçbir kısıt getirilmemiştir. Cümle içinde herhangi bir yerde herhangi bir zamanda değişken ataması yapılabilir. Bu da paralel programlamaya olağanüstü bir esneklik kazandırmaktadır. Fakat bu aşırı esnekliği sağlamak için çok karmaşık çoklu çevre (multiple environment) yönetimi gerekmektedir. Ayrıca bazı ciddi uygulama problemleri de mevcuttur [24] [28]. Bu zorlukların üstesinden gelebilmek için Flat Concurrent Prolog (FCP) tasarlanmıştır.

Bütün Flat CCND dillerinde olduğu gibi, FCP'de de, *guard* içinde sadece önceden tanıtılmış yüklemlere (predefined predicates) izin verilmektedir. Fakat FCP'nin *guard* kısmında değişken atamaya izin verilir. *Guard* hesaplama paralel değil sıralı olarak yapılır. Bir *fail* durumunda *guard* kısmında geriye-iz-sürme yapılır.

FCP'de *guard* çağrıları, bütün prosesi askıya alabilir. Bu durumda *guard* hesaplama sırasında yapılan bütün atamalar geriye alınır. Prosesin askıya alınmasına sebep olan değişken, prosese ulaştığında bütün *guard* hesabı yeni baştan yapılır.

Guard işlemleri sırasında iki çeşit birleştirme uygulanır: Atomik ve atomik olmayan birleştirme (atomic and non-atomic unification) Atomik birleştirmede, *guard* hesaplama sırasında, diğer prosesler *guard* tarafından hesaplanan değişkenlere asla ulaşamazlar. Diğer prosesler bu değişkenlere asla değer atayamazlar.

FCP'de *guard* kısmında hem atomik hem de atomik olmayan birleştirme mevcuttur.

Bu tezin başlamasına sebep olan ve ilham alınan esas kaynak Shapiro'nun CP [20] sistemidir. Shapiro CP sisteminin temellerini Kahn ve MacQueen'e [40] kadar götürmektedir. Kahn ve MacQueen bu makalesinde concurrent programlamanın akışlar ile modellenebileceğini göstermiştir. Daha sonra Van Emden ve deLucena [41] bu modeli mantık programlamaya uygulamışlardır. Clark ve Gregory [42] ise Van Emden ve deLucena'nın çalışmasını nondeterminism ve *guard* cümlesi kavramlarını kullanarak genişletmiştir. Daha sonra Shapiro, Clark ve Gregory'nin çalışmasını genişletmiş ve daha da açık (clear) bir hale getirerek, 1983 yılında Concurrent Prolog [20] sistemini tanıtmıştır. CP ile Clark ve Gregory'nin dilleri arasındaki farklar Shapironun makalesinde [20] ayrıntılarıyla verilmektedir.

2.7.2.3. Guarded Horn Clause

Guarded Horn Clause (GHC) [24] [25] [26], CP'ye bir alternatif olarak geliştirilmiştir. GHC'de değişken erişimini kısıtlamak için doğrudan veya açık bir biçimde takı kullanılmaz. Baş birleştirmesi sırasında bir değişkenle karşılaşıldığında veya *guard* işleme sırasında başta bulunan bir değişkene rasgelindiğinde, değişken değeri belli olana kadar bütün proses askıya alınır.

Başta bulunan bir değişkene ancak cümle seçildikten sonra bir terim ataması yapılır. Fakat GHC'de, [23]'de belirtilen çok ince bir problem ortaya çıkmaktadır. Birleştirme işlemi, bir değişkenin *guard* tarafında mı yoksa gövde tarafında mı geçtiğini tespit etmek zorundadır. Bunun tespiti de ancak ve ancak yürütme zamanı sırasında gerçekleşebilir. Bu da çok büyük bir performans kaybına sebep olur. Bu ince problemden dolayı GHC'nin sadece *flat* sürümü kullanılmaktadır.

Flat GHC'de, *guard* kısmında yine sadece önceden tanıtılmış yüklemlere izin verilir. Bütün *guard* işlemleri sıralı yürütülür. Eğer *guard* işleme sırasında atanmamış değişkenle karşılaşılırsa, bütün proses yine askıya alınır. Flat GHC'nin

bir diğer ismi ise KL1'dir.[11] KL1 sistemi aynı zamanda bir GC'ye sahiptir. CCND dillerinde geriye-iz-sürme kavramı olmadığı için farklı bir GC mekanizması mevcuttur.

Flat CCND dilleri arasında Flat Parlog atomic olmayan, FCP ise atomik birleştirme tekniğini kullanır.

Ayrıca Flat GHC, *guard* yürütmesi sırasında hiç bir değişkene değer atanmamasını temin eder.

2.7.3. Uygulama Modelleri

Bu bölümde, klasik CCND dillerinin uygulama modelleriden bahsedilecektir.

2.7.3.1. Multi-PSI Modeli

Japonlar Beşinci Kuşak Bilgisayar Projesi kapsamında PSI (The Personal Sequential Inference Machine) [31] adını verdikleri bir bilgisayar sistemi geliştirdiler. Daha sonra KL1 dilini üzerinde çalıştıran ve dağıtık bir mimariye sahip olan bir sistem [30] [32] geliştirildi. Kendi sistemimiz ile olan büyük benzerlikleri dolayısı ile bu sistemin temel tasarım prensiplerini burada vermek istiyoruz.

Her bir işlem elemanına (processing element) bir adet özel numara verilmiştir. Bu numara hem prosesler arası iletişimde hem de veri referanslarında kullanılacaktır.

Bütün işlem elemanları yüksek hızlı bir ağ (network) ile birbirlerine bağlanmışlardır.

Her bir işlem elemanının yerel bir adres uzayı (local adress space) mevcutur. Fakat her bir işlem elemanı uzaktaki verilere referans mekanizması ile ulaşabilir.

[11] Flat GHC = KL1, Kernel Language 1

Bütün proses elemanları birbirleri ile ağ üzerinden ileti (message) aktarımı ile haberleşirler.

Bütün uzak çağrılar (external calls) *meta call* kavramı yardımı ile yapılır. Bir *meta call*, call/3 iletisinden oluşur ve *call(Goal, Result, Control)* yapısına sahiptir. *Control* değişkenine *suspend, resume* veya *stop* değerleri atanır.

Uzak değişkenler (external references), aşağıdaki gibi, 3 elemanlı bir veri yapısı ile gösterilirler.

< *read flag, işlem elemanı numarası, cell identification number* >

Guard çağrıları sadece test yüklemlerinden oluşur. Bu tür birleştirme işlemine pasif birleştirme (passive unification) denir. Diğer bir deyişle *guard* işlemleri sırasında pasif birleştirme, gövde çağrılarının yürütülmesi sırasında ise aktif birleştirme (active unification) yapılır.

Giriş değişkenleri kesinlikle dışarından okunur, asla *guard* tarafında atanmaz. Değişkenleri sadece aktif birleştirme atar.

Guard çağrıları, uzak değişkenleri sadece okuyabilir, atayamazlar.

Uzak değişkenler aşağıdaki gibi okunur.

- *Guard* çağrıları yürütülürken uzak bir değişkene denk gelinirse *guard* işleme eylemi son *guard* çağrısına kadar devam eder. Bütün *guard* çağrıları işlendikten sonra, talep edilecek bütün değişkenler askı yığınına (suspension stack) alınırlar. Böylece bu *guard*'ı yürütmek için gerekli olacak bütün değişkenler bir kerede yığına atılır.
- Eğer askı yığınında eleman varsa, hemen değer-oku (read-value) iletisi üretilir ve değerler elde edilir.
- Bütün değerler elde edildiğinde, eğer ilgili cümle hala seçilmemiş ise, *guard* tekrar, baştan itibaren yürütülür.

Aktif birleştirme işinde, her zaman, birden fazla işlem elemanının aynı ortak değişkeni tutma olasılığı mevcuttur. Bu problemi engellemek için bütün işlem elemanlarına bir öncelik numarası verilmiştir (seniority rule). Böylece, küçük numaralı olan işlem elemanı, büyük numaralı bir işlem elemanı tarafından tutulan bir değişkeni elde edemez. Büyük numaralı işlem elemanının değişkeni bırakmasını bekler ya da bu değişkene bir terim atandığında değişken değerini elde eder.

2.7.3.2. Flat Parlog Modeli

Bu modelin özellikleri aşağıda kısaca verilmiştir.

İşlem elemanları yine bir ağ üzerinden birbirleri ile bağlıdırlar.

Bütün işlem elemanlarının kendine has yerel bellekleri mevcuttur.

Global bir bellek mevcut değildir.

Bütün işlem elemanları ileti aktarımı ile haberleşirler.

Eğer *guard* işleme sırasında uzak bir değişkene denk gelinirse bütün proses askıya alınır.

Bütün giriş değişkenleri gelene kadar proses askıda kalır. Gurad'da geçen bütün giriş değişkenlerinin eksiksiz olarak gelmiş olması gereklidir. Sadece == =/= gibi birkaç özel yüklem, giriş değişkeni gelmeden de işlem yapabilir. Bütün giriş değişkenleri atandıktan sonra yapılan birleştirme işine *strict* test denir. Atanmamış değişkenler üzerinde yapılan yürütmeye de *non-strict* test denir. Flat Parlog da her iki test şekli de mevcuttur.

Yine daha önce bahsedildiği gibi, çıkış değişkenlerine (output variables) ancak cümle seçildikten sonra (after commitment) atama yapılır.

Birleştirme sırasında uzak bir değişken ile karşılaşılırsa *unify* iletisi üretilir ve bu ileti ile değişkenin tutulduğu işlem elemanından, değişken talep edilir.

Veri akışı, yüklem seviyesinde (relation level) yapılan mod tanımlaması ile kısıtlanır.

2.7.3.3. Flat Concurrent Prolog (FCP) Modeli

FCP modelinin [33] [34] [35] özellikleri aşağıda kısaca özetlemiştir.

FCP hesaplama modeli, birleştirme, seçme ve indirgeme (unification, selection and reduction) adımlarından oluşur.

Guard işleme sırasında atomik birleşme özelliği vardır. Diğer hiç bir proses, *guard* değişkenlerine erişemez.

Birden fazla proses ortak bir değişkene erişmeye çalıştığı zaman ölümcül kilitlenme olasılığı ortaya çıkar. Bu problem de yine her bir işlem elemanına bir öncelik verilerek çözülmüştür.

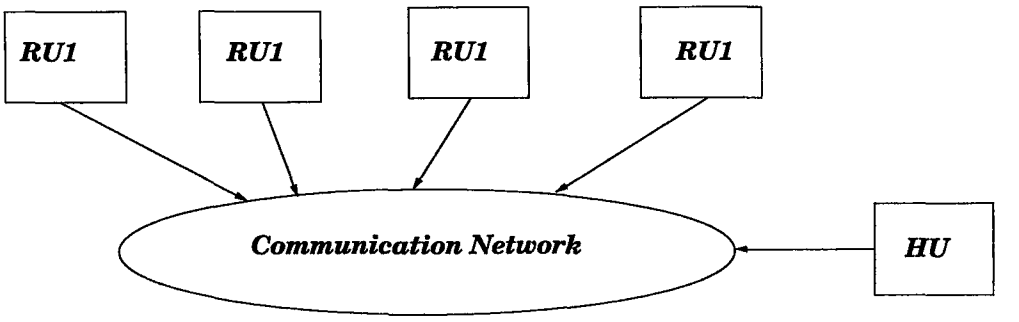
Mevcut her değişkenden bütün sistemde sadece ve sadece bir adet olması temin edilmiştir.

Global adres uzayı mevcuttur.

Guard hesaplama her zaman sıralı olarak yapılır.

Guard çağrıları sırasında uzak bir değişkene denk gelinirse bütün proses askıya alınır. Diğer *flat* dillerde de aynı özellik mevcuttur.

Baş birleştirmesi (head unification) *guard* işlemleri ile birlikte yapılır. Diğer bir deyişle baş birleştirmesi için cümlenin seçilmesi gerekmemektedir (head output unification). *Flat* dilleri içinde sadece FCP'de bu özellik mevcuttur.



RU : Reduction Unit
HU : Host Unit

Şekil 2.2: FCP sisteminin soyut yapısı

İndirgeme sırasında kullanılacak bütün değişkenler yerel makinede olmak zorundadır. Bundan dolayı üzerine yazım yapılacak bütün değişkenler yerel makineye taşınmak zorundadır. Bir değişkeni uzak bir makineden yerel bir makineye taşıma eylemine değişken göçü (variable migration) denir. Ölümcül kilitlenmeyi engellemek için bütün işlem elemanlarına sabit bir öncelik numarası verilmiştir.

Uzakta bulunan giriş değişkenleri gerekli olduğunda bütün proses askıya alınır.

Bütün Flat CCND dilleri arasında en karmaşık olanı FCP'dir. Fakat bütün akışlı paralel diller FCP içine gömülebilir [22].

FCP modelinin son derece basit senkronizsayon modeli mevcuttur. Senkranizasyon ne kadar basit ise uygulama o kadar kolay yapılır.

FCP'nin diğer bazı özellikleri aşağıdaki bölümde ayrıca incelenecektir.

2.7.3.4. FCP'nin bir transputer uygulaması

FCP'nin daha yeni bir uygulaması [36]'da tanıtılmaktadır. Bu yürütme modeli, 50 adet işlemcisi bulunan bir transputer sistemi üzerinde uygulanmıştır. Şekil 2.2'de bu sistemin soyut yapısı verilmiştir. FCP'nin bu uygulaması bizim önerdiğimiz sisteme son derece benzediğinden üzerinde ayrıntılı olarak durulacaktır.

FCP sistemi, sıralı FCP sistemlerinin bir ağ üzerinde birleşmesinden oluşmaktadır. Her bir sıralı FCP makinesine, indirgeme birimi (reduction unit) denir. Bu sistem paralel C ile yazılmıştır.

Ortak bellek veya paylaşılan bellek (shared memory) mevcut değildir. Bundan dolayı indirgeme birimleri arasındaki iletişim (interaction) ileti aktarımı ile sağlanır.

FCP sistemi ölçeklenebilir özelliğe sahiptir. Ağ topolojisi ve büyüklüğünden bağımsızdır.

Prosesler arası iletişim ve senkronizasyon ortak değişkenler ile sağlanır. Veri akışı modeli gereği, giriş değişkenlerinin tamamı belli olmadan bir proses ayağa kalkmaz, askıda bekler. Bir proses $p(A_1, A_2, \ldots, A_k)$ biçimindeki çağrı atomu (goal atom) ile temsil edilir. Burada k harfi, *arity* değerini gösterir. Bütün mantık programlama dillerinde olduğu gibi buradaki A_i argümanları da, bu prosesin veri durumunu (data state of the current process) gösterir.

Bir indirgeme işlemi aşağıdaki gibi yapılır.

P, herhangi bir mantık programı olsun. İçinde, yüklemlere ait pek çok cümle bulunsun. Bu yüklemlerden birini temsili olarak $C^{p/k}$ sembolü[12] ile gösterelim. Bu durumda bir p/k yüklemi aşağıdaki ifadeyi sağlar.

$$C^{p/k} = \{C_1^{p/k}, C_2^{p/k}, C_3^{p/k}, C_1^{p/k}\} \subseteq P$$

Buradaki her bir $C_i^{p/k} \in C^{p/k}$ cümlesi, bir tekrar-yazım (rewrite) kuralını temsil eder ve *guarded Horn cümlesi* (guarded Horn clause) yapısına sahiptir. $p(A_1, A_2, \ldots, A_k)$ prosesinin davranışını $C^{p/k}$ kümesinin sonlu bir alt kümesi belirler. *Guarded Horn* yapısına sahip her bir C_i cümlesi 2.1'deki gibi gösterilir.

Cümle başı ve bütün *guard* çağrıları işlendikten sonra cümle seçimi yapılır. Cümle seçildikten sonra her bir B_i elemanı için bir proses açılır. Aynı zamanda elde edilen yeni değişkenlere ve proseslere göre global ağ durumu (global network state) güncellenir.

$p(A_1, A_2, \ldots, A_k)$ çağrısı için açılan prosesde, $\{C_1^{p/k}, C_2^{p/k}, C_3^{p/k}, C_1^{p/k}\}$ cümleleri paralel olarak taranır. Baş ve *guard* çağrıları başarı ile işlendikten sonra seçme operatörü işlenir. İndirgeme amacı ile, p/k çağrısı için ilgili bir C_s cümlesi seçilmiştir. Bu seçimden sonra gövde üzerinde bulunan her bir B_i çağrısı için paralel bir proses açılır, artık indirgeme işlemi başlamıştır. *Non-determinism* şekli olarak yine *don't care nondeterminism* kullanılmıştır. Diğer bir deyişle indirgeme amacı ile seçilen ilk cümleden sonra diğer bütün alternatif çözümler gözardı edilecek, dikkate alınmayacaktır.

Bir prosesin davranışı indirgenecek cümlenin yapısına bağlıdır. Cümle yapıları daha önce 2.2, 2.3 ve 2.4'de verilmişti. 2.2'de verilen cümle yapısında, indirgeme eylemi her bir gövde elemanı için proses açar. 2.3'de verilen cümle yapısında sadece bir durum geçişi[13] (state transition) söz konusudur. 2.4'de verilen birim cümle yapısında ise bir sonlanma (termination) söz konusudur.

[12]Kısaltmaların anlamları: p: predicate, k: arity, p/k: p adındaki yüklem, C: clause, P: logic program.

[13]Mantık programlamada, prosesin durumu argümanlar tarafından tespit edilmektedir.

Daha önce de bahsedildiği gibi CCND dillerinin karakteristiklerini *guard* işlemleri tespit eder. *Guard* işlemlerinde iki yaklaşım vardır. Birinci yaklaşımda *guard* tarafında her türlü işlem yapılabilir [37]. Bu genel yaklaşımı CP, GHC ve Parlog uygulamaktadır. Diğer yaklışım ise *guard* çağrılarına kısıt getirmektir. Daha önce de bahsedildiği gibi bu tür dillere de Flat CCND dilleri denmektedir. Flat dillerinin en büyük özelliği, kontrol karmaşıklığını azaltması, dili uygulamanın daha basit olmasıdır. Fakat diğer taraftan *guard* kısmında sadece, daha önceden tanıtılmış yüklemler kullanılabildiği için programlama esnekliği azalmaktadır. Flat diller programlama kolaylığı ile uygulama zorluğu arasında uygun bir çözüm bulmaktadır. Yukarıda bahsi geçen bütün paralel dillerin ayrıca *flat* sürümleri de mevcuttur.

Buradaki FCP sisteminde yine klasik veri-akışı senkronizasyonu kullanılır. Diğer bir deyişle bütün giriş değişkenleri tamam olmadan proses yürütülmez (process suspension mechanism). Yine klasik olarak indirgeme sırasında atanmamış bir değişken mevcutsa bu proses, bu değişken belli olana kadar askıya alınır. Bu işleme indirgemeyi geciktirme mekanizması (reduction delaying mechanism) denir. FCP yine klasik olarak bu geciktirmeyi salt-okunur değişken kavramına dayanarak yapar. IPC, yine ortak değişkenler üzerinden yapılır.

Ve operatörü ile bağlı çağrılar ortak değişkenler üzerinde bir iletişim kanalı kurarlar. "p(X), q(X?)" gibi iki çağrıda, p/1 ve q/1 prosesleri arasında, X değişkeni üzerinden bir iletişim kanalı (communication channel) kurulur. İletişimin yönü her zaman X'den X?'ye doğrudur. p/1 tarafından üretilen X değeri, q/1 tarafından kullanılır ve tüketilir. Bu iletişimin daha genel hali Şekil 2.1'de verilmiştir. Burada p/1 tarafında bulunan X değişkeni çıkış kanalı (output channel), q/1 tarafında bulunan X? değişkeni ise giriş kanalı (input channel) gibi çalışır.

Buradaki hesaplama şekli, yine birleştirme, cümle seçimi ve indirgeme aşamalarından oluşur. Özel bir durum mevcut değildir.

Bir FCP programının yürütülürken yapılan ana işlem (central operation) proses indirgemedir (process reduction).

Cümle seçimi, yükleme ait olan bütün cümlelerin paralel olarak taranması ile yapılır. *Guard* tarafında sadece test yüklemleri bulunur. Bundan dolayı *guard* son derece hızlı bir biçimde işlenir. Esas zaman baş birleştirmesinde harcanır.

Her bir indirgeme ünitesi, global bellekteki bir veriye erişebilir. Bu FCP uygulamasında, fiziksel dağıtık bir mimari kullanılmıştır. Paylaşılan, mantıksal bir bellek mevcuttur. Verinin fiziksel olarak dağılmış olması uygulama programını etkilemez (transparent). Her bir indirgeme biriminde bulunan yerel adres uzayları birleştirilerek global bir adres uzayı oluşturulmuştur. Bir global adres uzayındaki değişken indirgeme birimi numarası ve yerel bir adresten meydana gelir. Bu adresi sembolik olarak (i, j) ile gösterelim. Bu gösterim tarzı Multi-PSI sisteminde bulunan gösterim tarzının daha basit bir halidir. Multi-PSI sisteminde ayrıca her değişkene bir de okuma biti eklenmektedir.

Bir indirgeme biriminde hesaplanan bir terim bütün indirgeme birimlerine kopyalanır. Eğer kopyalanan bir terim içinde değişken mevcut ise, bu değişken (i, j) adresi ile değiştirilir.

Mantıksal değişkenler dağıtık bir ortamda bulunurlar. Dağıtık ortamdaki değişkenler üzerinde çalışmak için aşağıdaki kurallar uygulanır.

- İki veya daha fazla indirgeme birimi aynı değişkeni güncellemek istediğinde karşılıklı dışlama (mutually exclusive write access) garantilenmelidir.
- Atama (binding) işlemi atomik olmalıdır. Diğer bir deyişle baş veya *guard* işlenirken bir başarısızlık (failure) durumunda, değişkenlerle ilgili hiç bir iz kalmamalıdır. Başarı (succeed) durumunda ise dağıtık halde bulunan bütün değişkenler aynı şekilde etkilenmelidir.

Yazılabilir bir değişkene (writable variable) sahip bir indirgeme birimine değişken sahibi (variable owner) denir. Diğer durumdaki değişkenlere sahip indirgeme

birimlerine ise değişken üyesi (variable member) denir. Bir değişkeni ancak ve ancak değişken sahibi atayabilir. Değişken üyesi diğer indirgeme birimleri, bu değişkene erişmeye çalıştıkları an askı durumuna geçerler.

Uzak değişkenlere (external variables) erişildiğinde veya bir değişken atandığında ve bu değişkeni bekleyen salt-okunur değişkenler (read only variables) olduğunda, indirgeme birimleri arasında senkron bir iletişim kurulur. İndirgeme birimleri bu özel durumlar hariç tamamen asenkron bir biçimde çalışırlar.

i numaralı indirgeme birimi ile j numaralı indirgeme birimi arasında her zaman bir yol (path) mevcuttur. i numaralı indirgeme birimi değişken sahibi, j numaralı indirgeme birimi ise değişken üyesi olsunlar. Değişken üyesi indirgeme birimi, değişken sahibinden değişkeni talep eder. Bu talep i ve j numaralı indirgeme birimleri arasında kurulan bir yol tarafından değişken sahibine iletilir. Bu işleme değişken göçü denir. Daha önce de bahsedildiği gibi değişken göçünden amaç, değişken sahibi olmayan indirgeme birimini değişken sahibi yapmaktır, değişkeni yerel hale getirmektir. Böylece bütün sistem üzerinde bir değişken sadece ve sadece bir kez mevcut olmaktadır. Sonuçta bir değişkenin sadece ve sadece 1 adet sahibi bulunmaktadır.

Bu uygulamada proses indirgeme çevrimi, yine klasik olarak baş birleştirmesi ve *guard* hesaplama ile gövde çağrıları için yeni proses'ler açma adımlarından oluşmaktadır. Özel bir durum mevcut değildir.

Bir X değişkeni value(X) değerini alırsa, bu X değişkenini bekleyen[14] bütün indirgeme birimlerine bu değişken dağıtılmalıdır. i numaralı indirgeme birimini RU_i ile temsil edelim. Bu indirgeme birimi X değişkenine bir atama yapmış olsun. $RU_{j1}, RU_{j1}, \ldots, RU_{jk}$ indirgeme birimleri de bu X değişkenini bekliyor olsunlar. Her zaman RU_i ile RU_j indirgeme birimleri arasında bir yol vardır.

Bir adet değişkene her zaman birden fazla proses yazmaya çalışabilir. Bu durumu kontrol eden mekanizma, senkronizasyonu sağlamak için bütün sistemi

[14]Diğer bir deyişle, kendi içinde bu değişkeni X? şeklinde barındıran.

ölümcül veya yaşayan kilitlenmeye[15] götürebilir [38]. Ölümcül kilitlenme problemi daha önce de anlatıldığı gibi bütün indirgeme birimlerine birer öncelik numarası verilerek çözülmüştür.

Sistem bazında (global state) durum değerlendirmesi, bütün indirgeme birimlerine aynı anda bir mesaj yaymakla yapılır. Bütün birimler belirli bir çevrim içinde bu mesaja yanıt verirler. Bütün sistemin askıya mı alındığı yoksa hesaplamanın başarı ile mi bittiği gelen yanıtların değerlendirilmesi ile anlaşılır.

2.8. Berk Prolog: Genel Tanıtım

Bu tez çalışmasında Prolog programlama dilini, paralel mantık programlama diline genişlettik (an extension of Prolog). Elde edilen amaç dil, CCND dil ailesinin bir üyesidir. Bu yeni üyeyi Berk Prolog[16] olarak adlandırdık.

Bu tezde sadece yeni bir programlama dilinin özellikleri ve kuralları verilmemiş aynı zamanda bu dilin dağıtık bir ortamda yürütülmesini sağlayan derleyicisi de yazılmıştır. Paralel derleyiciyi yazabilmek için GNU Prolog [44] sıralı derleyicisi kullanılmıştır. Gerekli kütüphaneler (libraries) yazılarak, GNU Prolog derleyicisi paralel Prolog programlarını dağıtık bir ortamda yürütebilecek bir hale getirilmiştir.

Bu bölümde daha çok yaptığımız sistemin genel özelliklerinden ve CCND dilleriyle olan farklılıklarından bahsedeceğiz. Berk Prolog ve bunun yürütme modeli olan Berk Sistem'in ayrıntılı biçimde incelenmesi bundan sonraki bölümlere bırakılmıştır.

---

[15] Bir değişkenin belirli indirgeme birimleri arasında, çevrimsel olarak sonsuz bir biçimde göç etmesi.

[16] Berk kelimesi, burada sıfat olarak kullanılmıştır.

2.8.1. Özellikler

Berk Prolog bir CCND dilidir ve içinde *guarded Horn* cümlelerini barındırır. Berk Prolog'un söz-dizimi 2.1'de verilen ifadeye benzemekle birlikte bazı farklılıklara sahiptir. Bu faklılıklarla elde edilen yeni *guarded Horn* cümlesine, orjinal *guarded Horn* cümlesinin genişletilmesi (extension of guarded Horn clauses) diyeceğiz. Genişletilmiş *guarded Horn* cümlesinin özellikleri aşağıda verilmiştir.

$$H \ :- \ G_1, G_2, ..., G_n \mid B_1, B2, ..., B_m \ . \quad n, m \geq 0 \tag{2.5}$$

Sözdizimi 2.5'deki gibi tanıtılmıştır.[17] 2.5 ifadesi görüntüsü itbariyle, 2.1'deki yapıya benzemesine rağmen H, G ve B terimlerinin tanıtılmasında pek çok farklılık arzeder.

H, cümlenin baş kısmını temsil eden, f/n şeklinde, klasik bir Prolog terimidir[18]. Bu terimi biz @/2 operatörü ile genişletiyoruz. Baş kısım sadece H şeklinde olabileceği gibi $H@Alias$ şeklinde de olabilir. *Alias* değeri 2.5 cümlesinin (clause) hangi bilgisayar ortamında veya indirgeme biriminde indirgeneceğini tespit eder. *Alias* değeri uygun bir ad olabileceği gibi bir liste de olabilir.

Sistolik programlamada kullanılan Logical Occam [45] dilinde @/2 i ile proses ataması mevcuttur. Ayrıca PMS-Prolog [46] dilinde ise @/2 kullanarak yüklem dağıtımı, bizim önerdiğimiz anlamdan daha farklı bir biçimde mevcuttur. Biz Berk Prolog'a her iki özelliği de farklı bir biçimde ekledik. Ayrıca gövde çağrılarına koşul getirerek @/2 operatörü ile beraber Horn cümlesini genişlettik.

[17] $G_1, G_2, ..., G_n$ ifadesini kısaca G ile ve benzer şekilde $B_1, B_2, .., B_n$ ifadesini de kısaca B şeklinde göstereceğiz.

[18] Klasik bir Prolog terimi, $n \geq 0$ olmak üzere n adet argümana ve f adında bir *functor*'a sahiptir. Argümanlar ise ya değişken ya da yine f/n şeklinde birer terimdir.

Örnek 2.1 *p/0 yükleminin sadece iki makineye birden dağıtılması.*

```
p(X)@[r1, r2] :- G  | B.
```

Örnek 1.1'de, p/1 cümlesi, r1 ve r2 adlı makinelerde ortaya çıkacaktır. Sistemde bulunan diğer makineler veya indirgeme birimleri p/1 cümlesine sahip olmayacaklardır. Klasik CCND dillerinde bulunmayan bu özellik programlama esnekliği sağlamaktdır.

Cümlenin baş kısmında, CP'de olduğu gibi salt-okunur operatör mevcut değildir. Parlog'da olduğu gibi cümle seviyesinde mod tanımı da mevcut değildir. Biz hem mod tanımını hem de salt okunur değişken ekini iptal ederek baş kısmın tamamen bir Prolog terimi olmasını sağladık. Böylece sıralı bir Prolog derleyicisi, baş birleştirmesini (head unificiation), kendi birleştirme algoritmasında hiç bir değişiklik yapmadan kullanabilecektir.

Baş birleştirilmesi sırasında bir proses asla askıya alınmaz. Çünkü giriş değişkenleri hazır olmayan bir çağrı yapılamaz. CP'de giriş değişkenleri hazır olsa da olmasa da çağrı yapılır. Eğer bir salt-okunur değişkene denk gelinirse ve bu değişkenin değeri henüz mevcut değilse, çağrı ve dolayısı ile proses askıya alınır. Parlog'da ve GHC'de de aynı mantık silsilesi geçerlidir. Biz proses senkronizasyonunu baş veya *guard* kısmında değil gövde kısmında ve prosesi çağırmadan evvel yapmaktayız.

Baş kısmında @/2 operatörünün kullanılması çok ilginç sonuçlar doğurmaktadır. CCND dillerinin tamamında, tanımlanan bir P programı sonlu sayıda *guarded Horn* cümlelerinden kuruludur. Her bir proses bu cümleleri kullanarak indirgeme yapabilir. Berk Prolog'da ise bu kavram daha da genişletilmiş ve daha kontrollü bir cümle dağıtımı getirilmiştir. Bir proses indirgemesi yapılırken, indirgeme biriminde mutlaka o yükleme ait cümlelerin bulunması gerekmektedir. Kullanıcı veya planlayıcı program hangi indirgeme biriminde hangi cümlenin bulunduğunu bilmelidir.

Örnek 2.2 *p/0 yükleminin bütün makinelere dağıtılması.*

```
p :- G  | B.
```

Örnek 1.1'de p/0 cümlesi sadece ve sadece r1 ve r2 indirgeme birimlerinde ortaya çıkacaktır. Başka indirgeme birimi olsa bile p/0 yüklemi onlarda mevcut olmayacaktır. Kullanıcı, "?- q@r3." şeklinde bir sorgu verdiği zaman, sorgu hemen başarısız olacaktır. Çünkü r3 biriminde p/0 yüklemi mevcut değildir.

Eğer kullanıcı bütün indirgeme birimlerine, bütün cümleleri dağıtmak istiyorsa @/2 operatörünü kullanmamalıdır. p :- G | B. örneğinde, p/0 yüklemi istisnasız olarak bütün indirgeme birimlerine dağıtılacaktır. Aslında bazı yüklemleri bu şekilde kullanmak israf olacaktır. Örneğin bütün formatlı ekran çıkışlarının r9 makinesi üzerinden yapılmasını istiyorsak, bu çıkışı sağlayan yüklemi sadece r9 makinesine göndermemiz yeterli olacaktır. Diğer indirgeme birimlerinde hiç kullanılmayacak olan bir yüklemin bulunmasına gerek yoktur.

Bu tanım şekli ayrıca, aynı adlı yüklemlerin farklı makinelerde farklı bir biçimde gözükmesini de sağlar. Bu da çok ilginç bir özelliktir.

Örnek 2.3 *p/0 yükleminin 2 makineye birden dağıtılması.*

```
p@[r1, r2]   :- G1  | B1.
p@[r3]       :- G2  | B2.
```

Örnek 1.3'de verilen $p/0$ yüklemini gözönüne alalım. Bu tanımda $p/0$ yükleminin sistem bazında tekdüzeliği, homojenliği mevcut değildir. "$? - p@r1$." çağrısı ile "$? - p@r3$." çağrısı birbirlerinden tamamen farklı sonuçlar üretebilecektir.

Berk Sistem için bu özellik bir sakınca veya kısıt teşkil etmez, çünkü kullanıcı @/2 operatörünü kullanmadığı an homojen bir yüklem dağıtımına sahip olacaktır.

Guard kısmı yine klasik olarak, Ve operatörü ile birleşmiş, $p(A_1, A_2, \ldots, A_n)$ çağrılarından oluşmaktadır.

Berk Prolog *flat* bir dil değildir. Bundan dolayı guard çağrıları aynen CP'de olduğu gibi her zaman ve her yerde, her türlü değişkeni atayabilirler. Bu değişkenler baş veya gövde değişkeni de olabilir. Diğer bir deyişle Berk Sistemde *safe* olma gibi bir kavram mevcut değildir.

Flat diller, *guard*'da sadece önceden tanımlanmış yüklemlere izin verirler. CP'de ise *guard* için hiç bir kısıt yoktur. Biz şu ana kadar *guard* için önerilen kısıtlardan farklı olarak başka bir kısıt önermekteyiz. Bizim önerimiz de *guard* kısmında sadece klasik Prolog çağrılarına izin verilmesidir. Flat olmayan diğer CCND dillerinde olduğu gibi *guard* kısmında paralel Prolog yüklemleri değil sadece seri Prolog yüklemleri çağrılmaktadır.

Bu kısıt son derece basit bir ihtiyaçtan doğmuştur. Bu tez çalışmasına başlarken, bizim amacımız standard, sıralı bir Prolog derleyicisini kullanarak, üzerinde hiç bir değişiklik yapmadan[19], paralel bir Prolog sistemi yazmaktı. Böylece halen kullanımda bulunan herhangi bir sıralı Prolog derleyicisine bizim yazmış olduğumuz kütüphaneler eklenince paralel bir CCND dili elde edilmiş olacaktı.

Yukarıdaki paragraflarda bahsettiğimiz gibi, baş kısmının standart bir Prolog terimi olması ve *guard* kısmında da sadece standart Prolog çağrılarının yapılmasından dolayı, yukarida 2.5 ifadesinde verilen cümle, *guard* operatörüne kadar sıralı Prolog ile yürütülebilir.

Hemen bu kısıtlardan çok basit bazı sonuçlar çıkmaktadır. Baş veya *guard* terimlerinde bulunan değişkenlerde senkronizasyonu sağlamak için salt-okunur eki mevcut değildir. Bundan dolayı baş veya *guard* çağrıları hiç bir zaman bir prosesi askıya alamaz. Diğer bütün CCND dillerinde, sadece ve sadece baş veya *guard* çağrıları prosesi askıya alabilirler. Berk Prolog'da, *guard* içindeki çağrılar sıralı

[19]Sıralı derleyicinin kaynak kodunun elimizde olmadığı veya elimizde olsa bile üzerinde değişiklik yapabilecek kadar bilgi veya vaktimizin olmadığı varsayılmıştır.

Prolog tarafından işleneceği için, *guard* çağrılarının paralel yürütülmesi mümkün değildir. Bütün guard çağrıları sıralı yürütülür.

Klasik CCND dilleri sıralı Prolog'u dilin bir parçası olarak kabul etmezler. Sadece CP dilinin sıralı Prolog ile basit bir etkileşim mekanizması mevcuttur. Bu mekanizma da bizim önerdiğimiz gibi sıralı Prolog'u tümüyle içine alacak şekilde değildir.

Şu ana kadar anlattıklarımıza göre baş terimi ve *guard* çağrıları tamamı ile sıralı Prolog tarafından desteklenmektedir.

CP'de olduğu gibi baş birleştirmesi ve *guard* hesaplaması atomiktir. Bu işlemler sırasında hiç bir indirgeme birimi hesaplanan değişkenlere erişemez. Aynı zamanda cümle seçimi başarısız olursa (clause selection failure) bütün hesaplanan değişkenler geri alınır.

Biz bu tezde kavramsal olarak son derece farklı bir yaklaşım öneriyoruz. Klasik CCND dillerinde baş birleştirme (head unification) ve *guard* hesaplama (guard evaluation) sadece ve sadece cümle seçimi amacı ile kullanılmaktadır. Esas hesaplama gövde yüklemlerinin argümanları tarafından yapılmaktadır[20]. Biz ise asıl hesaplama kısmının *guard* tarafında sıralı Prolog ile yapılmasını ve gövde yüklemlerinin hesaplamadan ziyade prosesler arası iletişim ve senkronizasyon amacı ile kullanılmasını öneriyoruz.

Seçme operatörü (guard operator, commit operator) bilinen klasik anlamında kullanılmıştır. Eğer yürütme *guard* operatörüne kadar gelirse, bu çağrıya ait bütün alternatif çözümler iptal edilir. Diğer bir deyişle CCND dillerinin ortak özelliği olan *don't care nondeterminism* uygulanmıştır.

2.5 ifadesinde gövde kısmı, $B1, B2, ..., Bn$ şeklinde, Ve ile bağlanmış, terimlerden oluşmaktadır. Berk Prolog'da B terimleri, klasik atomik çağrılar olabilir. Böylece

[20]Prolog bir ilişkisel (relational) dildir. İlişkisel dillerde ise sonuçlar her zaman argümanlar yardımı ile elde edilir. CCND dillerinde de bu mantığın hakim olması son derece doğaldır.

Berk Prolog, *guarded Horn* cümle yapısını tam olarak içermektedir. Eğer isterse kullanıcı B_i çağrılarını, @/2 operatörü ve koşul koyarak genişletebilir.

Genişletilmiş B_i çağrılarının en genel hali 2.6'da verilmiştir.

$$f(A_1, A_2, A_3, \ldots, A_n)@Alias \;\; with \;\; Koşul \tag{2.6}$$

Standrat *guarded Horn* cümlesi gösterimi, @/2 ve with/2 operatörleri ile genişletilmiştir. Bu operatörlerin kullanımı tamamen çözülen probleme özgüdür.

Alias değişkeni bir indirgeme birimi ismi veya isim listesi olabilir. p(X)@[r1, r2, r3] with [need(X)] ifadesinde $p/1$ çağrısı aynı anda paralel olarak r1, r2 ve r3 adlı indirgeme birimlerinde işlenecektir. Fakat klasik CCND dillerinde olduğu gibi, bu çağrı önce yapılıp sonra da ilgili indirgeme biriminde askıya alınmaz. Berk Prolog'da çok basit bir kural vardır, eğer giriş değişkenleri mevcut değilse çağrı bekletilir, yapılmaz. Bundan dolayı indirgeme sırasında *guard* operatörüne gelene kadar hiçbir çağrı, prosesi askıya alamaz. Çünkü bütün giriş değişkenleri hazır olduktan sonra çağrı yapılır. Verilen örnekte, eğer X değişkeni atanmamış ise bu proses askıya alınır. X değişkeni, başka herhangi bir indirgeme elemanı tarafından veya yerel hesaplama ile atandıktan sonra p/1 çağrısı derhal r1, r2 ve r3 adlı indirgeme birimlerine gönderilirler. r1, r2 ve r3 indirgeme birimlerinde X değişkeni paralel olarak tüketilir, kullanılır.

Parlog'da cümle seviyesinde (relational level), CP'de salt-okunur ek ile, GHC'de ise *nonvar/var* denetimi ile yapılan prosesler arası senkronizasyon, Berk Prolog'da sadece gövde çağrılarında ve *with*/2 operatörünün içine eklenecek *need*/n* mesajları ile sağlanır. Berk Prolog'da *with*/2 operatörünün başka kullanım alanları da vardır. Bundan sonraki bölümlerde konu ayrıntılı bir biçimde incelenecektir.

with/2 operatörü içinde kullanılan bir diğer özellik de çıkış değişkenlerinin açık olarak kullanıcı tarafından belirtiliyor olmasıdır. CP'de sadece salt-okunur değişkenler kullanıcı tarafından tespit edilir, GHC'de kullanıcı hiçbir değişkene

müdahalede bulunmaz, hem giriş hem de çıkış değişkenleri kullanım yerine göre tespit edilir. Örtük paralelliğe en yakın sistem GHC'dir. Parlog'da ise bir üst seviyede hem giriş hem de çıkış argümanları kullanıcı tarafından tespit edilir. *mod/1* tanımı ile yapılan bu müdahalede, birleştirmenin yönü dolayısı ile değişkenlerin tipi tespit edilmektedir. Berk Prolog'da ise çıkış değişkenleri veya üretici değişkenler, açıkça *share** iletileri ile, çağrı seviyesinde tespit edilir. Berk Prolog bu kullanım şekli ile kullanıcıya bir miktar fazla yük bindirmekte fakat programlama esnekliğini artırmaktadır.

Klasik CNND dillerinin ortak özelliklerinde olduğu gibi, prosesler arası iletişim yine akışlar vasıtası ile sağlanır. Prosesler arası senkronizasyon *need** iletisinin açık olarak belirtilmesi ile, çıkış değişkenlerinin tespiti ise *share** iletisi ile sağlanır.

Bu konudaki örnekler ilerleyen bölümlerde verilecektir. Şimdi, 1.4'de vereceğimiz basit örnek yukarıda anlatılan pek çok kavramı özetlemektedir:

Örnek 2.4 *@/2 ve with/2 operatörlerinin kullanımı.*

```
p@[r1] :-  q(X)@r2 with share(X),
           r(Y)@r3 with share(Y),
           r(X, Y)@[r4, r5] with need([X, Y]).

main :-    p@r1.
```

Berk Prolog'da ilk giriş noktası her zaman *main*/0 yüklemidir. Bu örnek program Berk Sistem'e yüklendiğinde $p/1$ yüklemi, "p@[$r1$] : $-\ldots$" tanımından dolayı derhal $r1$ adlı indirgeme birimine aktarılacaktır.

main/0 yüklemi çağrıldığı zaman, p@$r1$ teriminden dolayı, $r1$ adlı indirgeme birimi $p/0$ yüklemini çalıştırmaya başlayacaktır. Planlayıcı (scheduler) aşağıdaki işleri sıra ile yapar.

- $q(X)@r2$, ifadesinden dolayı, r2 makinesine q/1'i indirgemesini söyler,
- $r(X)@r3$, ifadesinden dolayı, r3 makinesine r/1'i indirgemesini söyler,
- $r(X,Y)@[r4,r5]$, ifadesinden dolayı, hem r4 hem de r5 makinelerine r/2'yi indirgemesini söyler.
- Aynı anda, paralel olarak r2, r3, r4 ve r5 adlı indirgeme birimleri ilgili çağrıları indirgemeye çalışır.
- Fakat r4 ve r5 adlı indirgeme birimleri X ve Y'ye ihtiyaç duydukları için indirgeme işine hiç girişmezler ve değişkenlerin kendilerine ulaşmasını beklerler.
- r2 ve r3 adlı birimlerde X ve Y aynı anda paralel olarak hesaplanır ve hesaplanan bu değişkenler derhal r4 ve r5 makinelerine gönderilir.
- r4 ve r5 birimlerindeki prosesler beklenen değişkenleri alınca ayağa kalkarlar ve X ile Y'yi tüketirler.
- Hesaplama bu şekilde sürer gider.

Kullanıcı, örnek 1.4'de verilen programı, aşağıda örnek 1.5'deki gibi de yazabilir.

Örnek 2.5 *Sadece with/2 operatörlerinin kullanımı.*

```
p :-    q(X) with share(X),
        r(Y) with share(Y),
        r(X, Y) with need([X, Y]).

main :- p.
```

Bu durumda $q/1$, $r/1$ ve $r/2$ çağrılarının hangi makinelerde indirgeneceği planlayıcının sorumluluğunda olacaktır. Planlayıcının yazımı bu tezin kapsamı dışında olup, ileriki çalışmalarda yapılacaktır.

Şu anda sistemde son derece basit bir planlayıcı mevcuttur. @/2 operatörü ile açık olarak verilmemiş olan bütün çağrılar yerel makinede indirgenirler.

Aslında Berk Sistem için planlayıcı yazımının çok büyük bir problem teşkil etmeyeceğini tahmin ediyoruz. Çünkü gövde çağrılarında bulunan @/2 operatörünün sağ çocuğu (right child) değişken olabilir. 1.6'da verilen örnek, Berk Sistem'de geçerli bir programdır.

Örnek 2.6 *Planlayıcının sisteme dahil edilmesi.*

```
p :-  schedule(R1, R2, R3)
      |
      q(X)@R1 with share(X),
      r(Y)@R2 with share(Y),
      r(X, Y)@R3 with need([X, Y]).
```

R1, R2 ve R3, henüz tespit edilmemiş olan indirgeme birimleri adlarıdır. *Guard*'da bulunan *schedule/n* programı, o anki sistem yüküne veye planlayıcı politikasına (scheduling policy) göre uygun indirgeme birimlerinin adlarını tespit edebilir. Tespit edilen indirgeme birimleri adları, *guard* operatöründen hemen sonra yerlerine atanır ve uygun dağıtma işlemi gerçeklenir. Ayrıca yukarıdaki yazım biçimi, kullanıcıdan tamamen bağımsız olarak, ön-işleme sırasında tespit edilebilir.

Şu anda Berk Sistem'de uygun bir planlayıcı olmadığından, 1.5'de verilen örnek program, ortamda birden fazla makine dahi olsa, tek bir makinede yürütülecektir.

Ayrıca Berk Sistemde çok ilginç bir özellik daha mevcuttur. Bir proses şartlı bir biçimde değişken talebinde bulunabilir. Örneğin *need_and([X, Y, Z])* mesajında, her üç değişken de aynı anda belli olmadan bu değişkeni bekleyen proses ayağa kaldırılmaz. Ya da, $need\_or([X, Y, Z])$ şeklindeki bir mesajda X, Y veya Z değişkenlerinden ilk önce hangisi değer almış ise bu değişkeni bekleyen proses ayağa kaldırılır. Bu yüzden $need\_or([X, Y, Z])$ ile $need\_or([Y, X, Z])$

birbirlerinden çok farklıdır. Yani *need_or* mesajının elemanları arasında sıraya bağımlılık mevcuttur. Bu ilginç özellik ile, kararlı bir sistem olan Berk Sistem'de *merge*/3 algoritması gerçekleştirilmiştir. Hiç bir CCND dilinde bu tür bir mantık yürütme mevcut değildir. Genelde yapılan uygulama, bir değişkeni bekleyen proseslerin tablosunun tutulması ve bir prosesi ayağa kaldıracak olan bütün değişkenler atandığında o prosesin ayağa kaldırılması şeklindedir. Bu uygulama Berk Prolog'daki *need_and* mantığıyla örtüşmektedir.

Berk Sistem'de, kullanıcının yazdığı paralel Prolog programları yürütme için uygun değildir. Bundan dolayı yazılan programlar Berk Sistem tarafından bir ön-işleme (pre-processing) aşamasından geçirilir.

Berk Sistem kararlı bir sistem olduğu için, *if-then-else* yapısı ve dolayısı ile *not*/1 yüklemi (negation as failure) problemsiz bir biçimde uygulanmıştır. Tabii burada *if-then-else*'deki *then* kısmı için bir kısıt mevcuttur. İleriki bölümlerde bu konulara değinilecektir.

FCP'de olduğu gibi, ortak değişkenlerin bütün sistemde mutlaka bir adet olmasına gerek yoktur. Her indirgeme birimi kullanacağı her bir değişkenin bir kopyasını bulundurabileceği gibi üretici değişkenleri de atayabilir. Bu kabullerden dolayı *değişken göç veya ölümcül/yaşamsal kilitlenme* (deadlock/livelock) gibi problemler Berk Sistem'de ortaya çıkmayacaktır.

Berk Sistem'de ortak değişkenin birden fazla üreticisi olabilir. Eğer üretilen ortak değişkenler paylaşılıyorsa, basit bir çakışma denetimi yapılarak sistemin bütünlüğü korunmaya çalışılır.

Yine burada CCND dilleri içinde bulunmayan bir kavram sunmak istiyoruz. Paralel mantık dillerinde, eğer *Ve* ile bağlı çağrılarda ortak değişken varsa bu değişkene paylaşılan değişken (shared variable) denir. Bağımlı *Ve* paralelliği ve prosesler arası iletişim zaten bu değişkenler yardımı ile yapılır. Fakat biz klasik anlamda tanıtılan bu tür değişkenlere paylaşılan değil ortak değişken diyeceğiz. Eğer ortak değişken prosesler arasında paylaşılıyorsa buna da paylaşılan değişken

diyeceğiz. Diğer bir deyişle Berk Prolog tanımında bir değişkenin paylaşılan olması için ortak olması yani *Ve* ile bağlı çağrılar arasında geçmesi ve diğer prosesler ile birlikte açık olarak paylaşılması gereklidir. Diğer bir ifadeyle, *need* ile açık olarak paylaştırılmayan hiç bir değişken, ortak bile olsa, paylaşılan değildir.

Ayrıca Berk Prolog'a körleme bekleyiş denilen bir kavram ekledik. Bu kavramla, bir proses tarafından hesaplanan bir değişken belirli bir indirgeme birimine doğrudan yönlendirilmektedir. Değişkeni bekleyen indirgeme birimi, değişkeni beklediğine dair hiç bir yere bir mesaj göndermez. Değişkeni hesaplayan proses, bu değişkenin hangi indirgeme birimi tarafından beklendiğini bilir ve oraya gönderir. Bu daha çok uygulamaya yönelik bir kavramdır. Daha çok çıkış veya *log* bilgilerinin belirli bir makinede tutulması amacı ile kullanılabilir.

3. BERK PROLOG: AYRINTILI İNCELEME

Bu bölümde, yeni bir CCND dili olan Berk Prolog'un bazı özellikleri ayrıntılarıyla anlatılmaktadır. Berk Prolog, bazı kavramları ile orjinal Concurrent Prolog'a benzemesine rağmen temelde pek çok farklılığa sahiptir. Berk Prolog'da farklı bir söz-dizimi (syntax) ve yorum (semantic) önerilmektedir.

3.1. Berk Prolog'un Söz Dizimi

Berk Prolog'daki söz-dizimi (syntax) Shapiro'unun sistemindeki söz-dizimine şekil olarak benzemesine rağmen temelde çok büyük farklılıklar arzeder.

Klasik Concurrent Prolog'da bir cümlenin (clause) söz dizimi aşağdaki gibidir.

Head ← Guard | Body.

Klasik tanımda *Head* ve *Guard* ifadeleri, *functor(arg, ...)* yapısı veya *atomic* şeklindeki tek bir değerden oluşur. Argümanlar da aynı yapıya sahip olabilirler. Fakat bazı değişkenler, değişkenler arası senkronizasyonu sağlamak için "?" ile temsil edilen son-düzenli (postfix) bir operatör alabiliriler.

Berk Prolog'da değişkenler arası uyum gövdeye (body) eklenen *with/2* operatörü ile sağlanmıştır. Berk Prolog'un genel söz-dizimi en genel hali ile tablo 3.1'de verilmiştir.

Kullanıcı paralel program yazarken bu söz-dizimini kullanmak zorundadır. Önişlemci (prepocessor) bu kuralı *berk_fact* olgularına (facts) çevirir. berk_fact olgularının yapısı daha sonra ayrıntılı olarak incelencektir.

Berk Sistem'deki *commit* " | " operatörü yine bilinen klasik anlamında kullanılmıştır. *Head* hariç diğer bütün elemanlar seçimliktir. *Body* mevcut değilse *call(true)* olduğu kabul edilir.

Fakat burada *Head, Guard* ve *Body* içinde "*?*" operatörü bulunmaz. Değişkenler için her türlü tanım *ConditionList* içinde yapılır. Ayrıca değişkenler için sadece *read only* özelliği dışında başka özellikler de tanımlanabilmektedir. Berk Prolog cümleleri içinde bulunan bütün yapılar klasik Prolog'daki yapılar ile bire bir örtüşür, hiç bir farklılık mevcut değildir.

Başta (head) bulunan @/2 operatörü bu cümlenin hangi makineye taşınacağı konusunda ön-işlemciye bilgi verir.

Aliases değeri herhangi bir istemci adı veya istemci adları listesi olabilir. Örneğin, hiç bir agümanı olmayan p/0 yüklemini ele alalım. Bu yüklem aynı zamanda hem *tornado* hem de *mig* adlı istemcilere yüklenecek olsun.

```
p@[tornado, mig] ← q(X) |
                   r(X, Y)@fulcrum,
                   t(Y)@[mig, fulcrum],
                   w(Y)@all.
```

Yukarıdaki örnekte, @/2 operatörünün hem baş hem de gövde içinde nasıl kullanılacağı gösterilmiştir. Head'de bulunan @/2 operatörü p/0 yükleminin hem *tornado* hem de *mig* adlı makinelere yükleneceğini garantiler. Diğer bir deyişle p/0 yüklemi, bu makinelere dağıtılır, gönderilir. Eğer sistemde başka istemciler varsa p/0 diğer işlemcilere yüklenmez. Bu yüklenme işlemi derleme zamanında yapılır, yürütme zamanında gözönüne alınmaz.

Eğer baş önünde @/2 operatörü bulunmazsa bu cümle (clause) sistemde bulunan bütün istemcilere gönderilir. Eğer sistemde dört adet istemci mevcutsa

```
p ← Guard | Body.
```

ifadesi açıkça şöyle yazılabilir.

p@[mig, tornado, phantom, fulcrum] ← Guard | Body.

şeklinde yazılabilir. Her iki yazım şekli de birbirlerine tamamen denktir. Eğer isim listesi olarak boş küme mevcut ise, uylaşım gereği ve programlama kolaylığından dolayı

p@[] ← Guard | Body.

ifadesi her zaman gözardı edilecek, dikkate alınmayacaktır. Diğer bir deyişle yukarıdaki gibi bir ifadenin bütünü doğrudan *true* kabul edilir ve bu cümlenin çözüme bir katkısı olmaz. Bu yazım şekli ile bir tür etkisiz eleman, *etkisiz yüklem* tanımı yapmış oluyoruz.

Gövde atomlarında da @/2 operatörü bulunabilir. Bunların anlam olarak *Head*'de bulunan @/2 operatörü ile hiç bir ortak yönü yoktur. En azından buradaki operatörler yürütme zamanında iş görür. Gövdede bulunan @/2 operatörü her bir atomik çağrının önüne gelir ve bu çağrının hangi makinelerde indirgeneceğini tespit eder.

r(X, Y)@fulcrum çağrısında, *r/2*'nin *fulcrum* adlı makinede indirgenmesi sağlanır. Bu bir *açık çağrı (explicit call)* şekildir. *r/2* yükleminin *fulcrum* adlı makinede daha önceden yüklenmiş olması gerekir. Berk Prolog tarafından varlık denetimi yapılmaz, eğer bir yüklemin bulunmadığı bir istemcide indirgeme yapılmaya çalışılırsa hesaplama (concurrent computation) hata ile biter (failure of the computation). Açık çağrıda bütün sorumluluk programcıya terkedilmiştir.

w(X)@all şeklindeki bir çağrı aynı anda bütün istemcilere gönderilir. Şekil 4.2'deki örnek sistemi gözönüne alırsak w(X)@[mig, tornado, phantom, fulcrum] ile w(X)@all çağrısı birbirlerine denktir.

Atom@[] şeklindeki bir çağrı yine uylaşım gereği doğrudan *call(true)* şeklinde işlem görür. Diğer ifadeyle hiç bir etkiye sahip değildir.

Gövde içinde her atomun @/2 operatörünü alması zorunlu değildir. Kullanıcı isterse çağrıları serbest bırakır. Bu durumda kapalı bir yazım şekli ortaya çıkar

Tablo 3.1: Berk Prolog'un söz dizimi

Head	:- Guard | Body .
Head	:- Body .
Head.	
Head	→ Atom
Head	→ Atom@Aliases
Atom	→ { functor(Args,...) şeklindeki klasik yapı. }
Aliases	→ AliasesList
Aliases	→ ClientName
AliasesList	→ [ClientName|AliasesList]
AliasesList	→ []
ClientName	→ { Bir istemci için verilmiş olan bir prolog atomu }
ClientName	→ all
Guard	→ Goal, Guard
Guard	→ Goal
Goal	→ { Klasik Prolog Çağrıları }
Body	→ Atom@BodyAliases with ConditionList
Body	→ Atom@BodyAliases
Body	→ Atom with ConditionList
Body	→ Atom
BodyAliases	→ Aliases
BodyAliases	→ all
Variables	→ Variable
Variables	→ VariableList
Variable	→ { Klasik Prolog Değişkeni }
VariableList	→ [Variable|Variables]
VariableList	→ []
ConditionList	→ share(Variables)
ConditionList	→ share(Variables) with Alises
ConditionList	→ need(Variables)
ConditionList	→ need_and(Variables)
ConditionList	→ need_or(Variables)
ConditionList	→ wait(Variables)
ConditionList	→ wait_and(Variables)
ConditionList	→ wait_or(Variables)

ve bu çağrının hangi istemcide indirgeneceği kararı Berk Sistem'in planlayıcısına (scheduler) terkedilmiş olur. Berk Sistem için şu anda bir planlayıcı mevcut değildir. Fakat eklenecek herhangi bir planlayıcı yöntemi sistemin genel mantığını değiştirmeyecektir. Geliştirme aşamasında yapılması gereken çalışmalardan biri de sisteme bir planlayıcının dahil edilmesidir. Planlayıcı çalışması ayrı bir tez konusu niteliğindedir.

Berk Sistem için son derece basit bir planlama yöntemi (scheduling) seçilmiştir. Açık olarak hangi istemcide indirgeneceği belirtilmeyen çağrılar o an hangi istemcide bulunuyorlarsa o istemcide indirgenirler.

Bu tür atomlardan, yerelde indirgenecek atomlar veya çağrılar, geri kalan diğer gövde atomlarından ise uzak (external) istemcilerde indirgenecek atomlar veya çağrılar olarak bahsedeceğiz. Eğer bir cümle baş birleştirmesi (head unification) ve *guard* hesaplamasından[1] geçerse, bu cümle seçilmiştir (selected clause). Seçilen cümlede yapılan temel işlemlerden birisi gövdede bulunan atomları yerel çağrılar ve uzak çağrılar olarak sınıflandırmaktır.

Bu tanımlamalara göre sadece gövde içindeki atomlar hem klasik Prolog hem de concurent Prolog çağrıları yapabiliriler.

Baş birleştirmesi için standard MGU (Most General Unifier) kullanılmıştır. Bütün standard Prolog sistemleri bu yöntemi kullandığından dolayı, Berk Sistem'e geçiş sırasında, orjinal Prolog sistemi üzerinde herhangi bir değişiklik yapılmasına gerek kalmayacaktır. Ayrıca *guard* içindeki ifadeler standard Prolog ifadeleridir ve soldan sağa doğru yürütülürler. Yerel çağrılar, uygulama kolaylığı açısından soldan sağa doğru, yazılan sırada işletilirler. Uzak çağrılar ise tamamen planlayıcının seçtiği sırada işletilirler.

Berk Sistem'de ana hesaplama parçası olarak *guard* seçilmiştir. Gövde ise daha çok prosesler arası iletişime uygundur.

---

[1]Biz, flat bir sistem kullanmadığımız için, *guard* içinde yapılan işlemlerden, test değil hesaplama olarak bahsedeceğiz.

Eğer atom@[mig, fulcrum] gibi açık çağrı mevcut ise atom, hem *mig* hem de *fulcrum* adlı makinelerde aynı anda, paralel olarak indirgenmeye çalışılır. Bu durumda değişkenlerin senkrenizasyonu kullanıcının sorumluluğundadır.

Guard ifadesi klasik Prolog atomlarından oluşur, hiç bir istisnası mevcut değildir.

@/2 operatörünün ve örnekte geçen diğer ifadelerin hangi anlama geldiği ilgili konu başlıkları altında ayrıca anlatılacaktır.

3.2. İndirgeme (Reduction)

Berk Sistem'de indirgeme sadece ilgili istemcide yapılır. Sunucu hiçbir zaman indirgeme işlemine müdahale etmez. İndirgenecek Atom ile ilgili cümleler, yukarıdan aşağıya, kullanıcının yazdığı sırada (text order) taranır. Berk Sistem bu şekildeki taramayı her zaman garanti eder. Bu tür sistemlere kararlı (stable) sistem denir. Berk Sistem kararlı bir sistemdir. Böylece *if-then-else* yapıları son derece kolay bir biçimde uygulanabilmektedir.

İndirgencek Atom, önce yazılan sırada yukarıdan aşağıya kontrol edilir. Head ile Atom bilinen klasik yolla birleştirilir. Eğer birleştirme işlemi başarılı olursa *guard* içindeki atomlar tek tek çağrılır. Bu işlem doğrudan ISO Prolog'un *call/1* yüklemi ile yapılır. Eğer *call(Guard)* çağrısı başarılı olursa, *commit* operatörü devreye girer. Bu operatör klasik *cut* operatörü gibi çalışır ve bütün *alternatif çözümleri (choice points)* yok eder.

Yüklemin bu cümlesi, artık seçilmiştir. Eğer varsa, sonraki bütün cümleler gözardı edilir. *Guard* kesinlikle klasik Prolog çağrılarından oluşmaktadır ve *guard* içinden Berk Prolog yüklemleri çağrılamaz.

Shapiro'nun sistemi [20] ile Berk Sistem arasındaki en önemli farklardan biri *salt okunur* (read only) değişkenler için "*?*" operatörünün kullanılmamasıdır. İkinci önemli fark ise *guard* içinde Berk Prolog çağrılarının bulunmamasıdır. Guard

içinde salt okunur değişken ve Berk Prolog çağrısı mevcut olmadığı için *guard* içinde bulunduğu prosesi askı *(suspend)* durumuna geçiremez. Bu özellik ise Berk Prolog ile CP arasındaki üçüncü önemli farkı oluşturur. İşte bu üç önemli fark, CP ve Berk Prolog'u birbirinden kesin çizgilerle ayırmaktadır.

Guard içinde bütün klasik Prolog çağrıları bulunabileceği gibi *cut* kontrol operatörü de bulunabilir. Eğer Berk Sistem'in üzerinde çalıştığı klasik Prolog sisteminin *call/1* uygulaması, *cut* işlevini gerçeklerse, *guard* içinde bulunan cut karakteri de bilinen yan etkilerini gösterir. Diğer durumlarda ise bu kontrol operatörü göz ardı edilir. Diğer bir deyişle *guard*'ın davranışı, Berk Sistem'in yazıldığı klasik Prolog sistemindeki *call/1* çağrısının davranışına bağlıdır.

Gövde kısmı ise concurrency'nin asıl gerçekleştiği yerdir. Gövde atomları, *guard* atomlarından faklı olarak hem Berk Prolog hem de sıralı Prolog çağrıları yapabilirler.

Baş ve *guard* çağrıları her zaman soldan sağa doğru klasik Prolog'daki gibi işlenirler. Gövde kısmı concurent Prolog'a geçiş kısmıdır. Gövdenin yazım şekline ve planlamaya göre gövde içindeki çağrılar, yerel ve uzak olarak iki ayrı gruba ayrılırlar. Uzak çağrı yapan atomlar hemen sunucuya gönderilirler. Sunucu da atomları indirgenmek üzere ilgili istemcilere dağıtır.

Gövde içindeki her atom, *@/2* operatörü ile birlikte, aynı zamanda *with/2* operatörü de alabilir. *with/2* operatörü paylaşılacak olan değişkenleri tanımlar ve değişkenler arası senkronizasyonu sağlar. Örnekler üzerinden gösterecek olursak;

```
p ← q |
      r(X)@mig with need(X),
      s(X)@fulcrum with share(X).
```

r/1 çağrısı *mig* adlı makinede, *s/1* çağrısı ise *fulcrum* adlı makinede indirgenecektir. r(X) çağrısı *mig* makinesinde indirgenmeden evvel *need(X)* denetimine takılır. Burada X'in mutlaka *atanmış (bounded)* olması gereklidir.

Önce X'in *mig* içinde olup olmadığı kontrol edilir. Mevcut olmadığı için bu değişken sunucudan talep edilir.

Bir değişken sunucudan bir kez talep edilir. X değişkeni gelene kadar r(X) atomu kendini *askıya alır (suspend)* ve bekler. s(X) çağrısı ise *fulcrum*'da indirgenecektir. İndirgendikten sonra elde edilen X değeri hemen sunucuya gönderilir. Sunucu tarafında ise X'i bekleyen istemcilerin adları mevcuttur. Sunucu hemen kendine gönderilen X değerini *mig* adlı istemciye gönderir. *mig* istemcisi de gelen X değerini alarak, kendini askıdan kurtarır ve yürütme işlemine devam eder.

need/1 yerine *wait/1* yapısı da kullanılabilir. Bu durumda X değişkeninin beklendiği sunucuya bildirilmez. Bu özel durumu körleme bekleyiş olarak adlandıracağız. Bu durumda sunucu, kendine gelen değişkene kimin ihtiyacı olduğunu bilmez. Bu gibi durumlarda değişkeni hesaplayan istemci hesaplanan bu değişkeni ilgili istemciye göndermek için, değişken içine ayrıca bir alan ekler. Bu alan sayesinde sunucu kendine gelen değişkeni körleme bekleyiş yapan istemciye gönderir. Bu durum aşağıda örneklenmiştir.

```
p ← q |
        r(X)@mig with wait(X),
        s(X)@fulcrum with [share(X) with mig].
```

Bu durumda, r(X) çağrısı indirgeme sırasında wait(X)'den dolayı *askı* duruma geçecektir. Fakat X değişkeninin beklendiği sunucuya bildirilmez. Diğer deyişle sunucudan bir değişken talebinde bulunulmaz ve *kör* bir biçimde X değişkeninin kendine sunulmasını bekler. Diğer tarafta ise s(X) indirgenir ve X değeri elde edilir. Elde edilen X değeri hemen sunucuya gönderilir. Fakat değişken gönderilirken, değişkene ihtiyacı olan istemcilerin adları da beraber gönderilir. Sunucu bu değişkeni alır almaz hemen ilgili istemciye yollar.

wait/1 özelliği aynı zamanda yerel sistemde değişken bekleme durumunda da kullanılabilmektedir. Bu konu ileriki bölümlerde örneklerle açıklanacaktır.

wait/1 ile *need/1* arasındaki en önemli fark *wait/1*'in daha az ağ yükü yaratmasıdır. Çünkü *need/1* yapısı beklenen her değişken için bir adet *need* mesajı gönderir. *wait/1* yapısında ise buna gerek yoktur. Fakat *wait/1*'in kullanımı programcılık açısından daha fazla maharet gerektirir ve hatalı kullanıma daha müsaittir.

4. BERK SİSTEM

Bir paralel mantık programlama dili olan Berk Prolog'un TCP/IP ortamındaki yürütme modelini Berk Sistem olarak adlandırıyoruz. Bu bölümde Berk Sistem'in Prolog ile gerçekleştirilmesi üzerinde durulmaktadır.

Yütürme modelleri, örtük paralel sistemlerde WAM seviyesinde yapılan değişiklikler ile gerçeklenir [16] (clause level compilation).

Açık Paralel Prolog sistemleri çeşitli yazılım araçları ile kurulabilir. Örneğin KL1 sistemi C ile gerçekleştirilmiştir. Berk Prolog'un yürütme modelinde uygulama dili olarak sıralı Prolog tercih edilmiştir. Bu yürütme modelinde, standart Prolog sisteminin kendi yüklemleri kullanılarak paralel bir kütüphane yazılmıştır.

Bu paralel kütüphaneyi kendisine ekleyen her standart Prolog sistemi doğrudan paralel Prolog sistemine dönüşecektir.

Japonların Beşinci Kuşak Bilgisayar Projesi'nin başarılı olmamasının en büyük nedenlerinden biri, bizce standart Prolog sistemlerini göz ardı ederek kendi sistemlerini kurmaya çalışmalarıdır. Bu durumda klasik sistemlerde yapılan geliştirme ve genişletmelerden mahrum kalmışlardır.

Berk Sistem'de kullanılan paralelleştirme kütüphanesi klasik Prolog sisteminin yapısında herhangi bir değişiklik yapmamaktadır. Kütüphane doğrudan orijinal sistemin yüklemleri ile yazıldığıdan, bu sistemin kaynak kodlarına ulaşma problemi de olmayacaktır. Ayrıca orijinal sistemde bulunan, DCG, CLP gibi genişletmeler de doğrudan paralel sisteme geçmiş olacaktır.

Biz Berk Sistem'i kurarken standart Prolog sistemi olarak GNU Prolog'u seçtik.

4.1. Geliştirme Ortamı

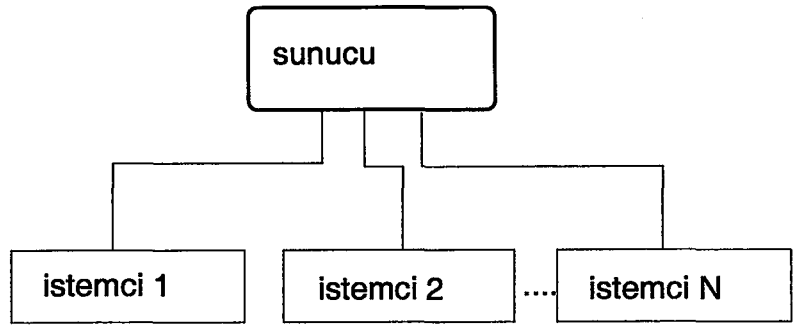
Berk Sistem'in geliştirme ortamı olarak Linux işletim sistemi kullanılmıştır. IP yığın (IP stack), Linux işletim sistemi ile beraber geldiği için elde edilen yürütülebilir kodlar bütün Linux sistemlerde kolayca çalışabilecektir.

Berk Sistem'in geliştirme dili olarak GNU Prolog kullanılmıştır. GNU Prolog dışında pek çok Prolog ile deneme yapılmış ama GNU Prolog'da karar kılınmıştır. Çünkü GNU/GPL lisansına sahiptir, geliştirme amaçlı kullanmak için hiçbir ücret talep edilmemektedir. Sürekli olarak geliştirilmektedir. Bir haber gurubu mevcuttur. İletilen problemler çok kısa sürede yanıtlanabilmektedir. Ayrıca hem yorumlayıcı (interpreter) hem de derleyici (compiler) olarak çalışabilmektedir. Her ne kadar kendi sistemimizde kullanmasak da son derece güçlü, iki yönlü C desteği mevcuttur. Ayrıca bu sistemi seçmemizin ana nedenlerinden biri unix soket kütüphanesini doğrudan Prolog içinde kullanabilmemizdir. Yine güçlü bir Giriş/Çıkış (Input/Output) desteği mevcuttur. Bütün G/Ç sistemi, unix'in stream tabanlı G/Ç sistemi ile aynı özelliklere sahiptir. Bu özellikler sistem içinde çokça kullanılmıştır. Berk Sistem'de hiç kullanılmamasına rağmen CLP (Constriant Logic Programming) desteği mevcuttur. İlerisi için böyle bir desteğin bulunması çok önemlidir. Bütün bunların yanında artık toplama (Garbage Collection) gibi standart Prolog özelliklerini de desteklemektedir.

4.2. Paralel Mimari Sınıfı

Berk Sistem MIMD (Multiple Instruction Multiple Data) tipinde bir paralelliğe sahiptir ve tam bir istemci/sunucu (client/server) uygulamasıdır. Bu özelliği ile *Beowulf* tipi sistemlerde çalışmaya son derece elverişlidir.

Bilgisayarlar arasında veri iletişimini sağlamak için PVM (Parallel Virtual Machine) veya MPI (Message Passing Interface) benzeri yardımcı kütüphaneler kullanılmamıştır. Bunların yerine, aynı makine üzerinde veya farklı makineler



Şekil 4.1: Berk Sistem'in topolojik yapısı

üzerinde çalışan prosesler, doğrudan unix'in en alt düzey soket kütüphanelerini (low-level socket library) kullanmaktadır. Bu sayede derlenmiş olan Berk Prolog programlarının yürütülebilir kodları IP yığın (IP stack) yüklü herhangi bir makinede yardımcı kütüphaneler, Prolog derleyicisi veya yorumlaycısı olmadan kullanabilmektedir. Böylece programların taşınabilirliği üst seviyelere getirilmiştir. İnternet ortamı veya *TCP/IP* ortamında bulunan Linux makinelerinde derlenmiş kodlar problemsiz olarak çalışabilmektedir.

4.3. Sistemin Topolojisi

Bir bilgisayarda Berk Sistem'in çalışabilmesi için IP yığın yüklü herhangi bir Linux makine olması yeterlidir. Berk Sistem ile derlenmiş kodun paralel çalışabilmesi için bir adet makine sunucu (server) en az bir adet makine de istemci (client) olarak seçilmek zorundadır. Sunucu görevini üstlenen makinenin ana işlevi indirgenecek olan atomları (atomic goal) ilgili makinelere göndermek ve değişken alışverişi için bir santral görevi görmektir.

Genel bağlantı yapısı şekil 4.1'deki gibidir. Mutlaka tam bir adet sunucu mevcuttur. En az 1 adet de istemci olmak zorundadır. Teorik olarak istemci sayısında bir sınır yoktur.

Şekil 4.1'deki çizim sadece lojik bir gösterimdir, fiziksel bağlantıları gerçek anlamda göstermez. Sunucu ve istemciler internet üzerinde keyfi olarak dağılmış herhangi bir makine olabileceği gibi, *Beowulf* mimarisine sahip, doğrudan *switch* üzerinden birbirlerine bağlanmış makineler de olabilirler. Berk Sistem'in çalışabilmesi için IP ortamında makinelerin birbirlerini görmesi yeterlidir.

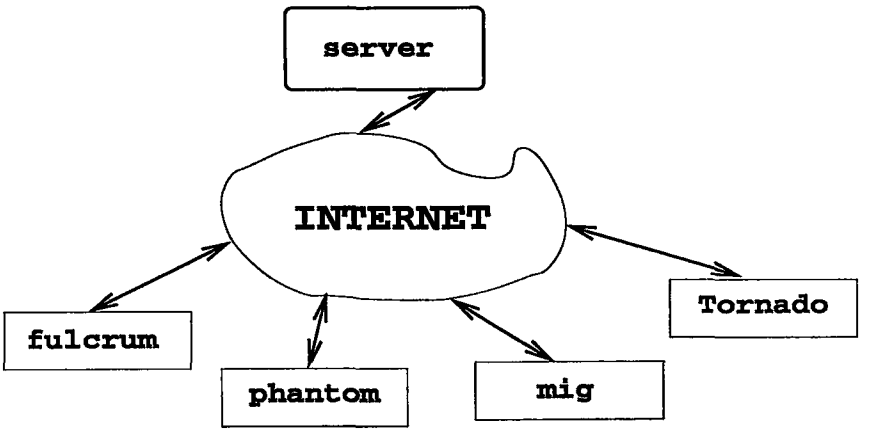
Programcı, her istemciye keyfi bir ad vermek zorundadır. Her istemcinin adı diğerlerinden farklı olmalıdır. Ayrıca hiçbir istemci adı *server* olamaz. Ayrıca *all* kelimesi de istemci adı olarak kullanılamaz. Çünkü bu ad bütün istemcileri temsil etmek için kullanılacaktır. Sunucunun adı olarak da her zaman *server* kelimesi kullanılacaktır.

Unix makinelerin domain isimleri ile istemcilere verilecek olan isimlerin hiçbir ilgisi yoktur. Fakat istenirse bu isimler de kullanılabilir. İstemci isimleri Prolog isim verme kurallarına uygun olmalıdır. Örneğin, bu isimler büyük harfle veya rakamla başlayamazlar.

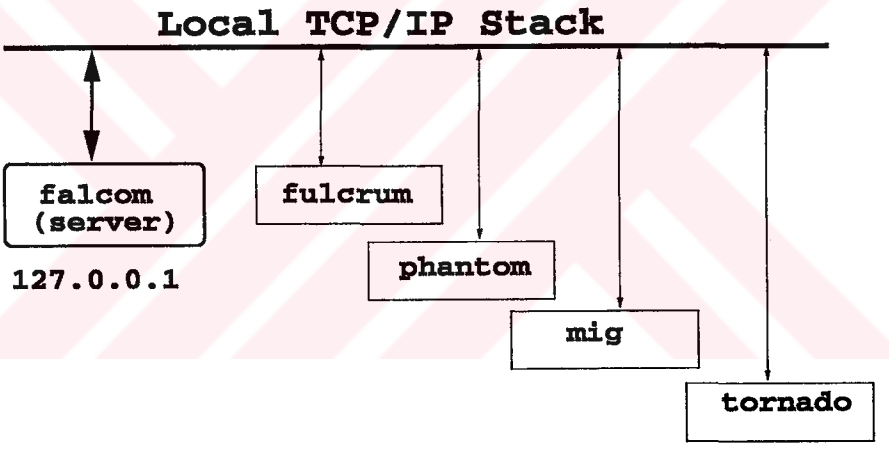
Örnek bir mantıksal yapı kuralım. Sistemde 4 adet istemci ve bir adet sunucu bulunsun. En az 1 sunucu ve en az 1 istemci olmak zorunda idi. Şunu da göz önünde bulundurmak gerekir ki bilgisayarların illa ki ağ (network) ortamında bulunması şart değildir.

IP yığın yüklü her Linux makine içinde *localhost* tanımı mevcuttur. Bu tanım sayesinde ağ olmadan, ağ varmış gibi çalışmak mümkün olmaktadır. localhost adlı makine hem sunucu hem de aynı zamanda bir istemci için ayrı bir makine gibi davranabilir. Yazılan programları test etmek için localhost ortamı çok ideal bir ortamdır. Teorik olarak böyle bir sistemde istenildiği kadar istemci tanımı yapılabilir, pratikte ise sistemin kaynakları ile sınırlıdır.

Dört adet istemci bulunan sistemde, önce istemcilere isim verilmelidir. Bu isimler, örneğimiz için sıra ile *fulcrum*, *phantom mig* ve *tornado* olsunlar. Sunucunun adı ise her zaman *server* olarak önceden tespit edilmiştir. Kullanıcı sunucu için ayrı bir adlandırma, isim tahsisi yapamaz.



Şekil 4.2: İnternet ortamında adlandırma



Şekil 4.3: Localhost ortamında adlandırma

İnternet ortamında ve bir localhost sisteminde örneğimizin genel görüntüsü Şekil 4.2 ve Şekil 4.3'deki gibi olacaktır. Şekillerden de kolayca tespit edilebileceği gibi istemcilerin sadece sunucunun IP bilgisini veya domain ismini bilmesi yeterlidir. Sunucuya, istemciler hakkında açıkça bilgi vermeye gerek yoktur. Çünkü istemciler bağlantı isteğinde bulunduğunda, sunucu bunu kabul eder ve hemen istemci ile arasında iki yönlü bir TCP/IP bağlantısı kurar. Bütün sistem TCP/IP üzerinde çalışmaktadır.

Berk Sistem'inin ayağa kalkıp indirgeme (reduction) yapabilmesi için bütün istemcilerin bağlantılarını tamamlamaları gerekmektedir. İndirgeme başlamadan

önce sunucu her bir istemcinin bağlantı kurmasını bekler. Bütün istemciler bağlantı kurduktan sonra sunucu indirgeme işlemine başlayabilecektir.

Herhangi bir istemci yürütme sırasında herhangi bir sebepten dolayı bağlantıyı keserse bütün sistem yani Berk Sistem'i çöker. Aynı zamanda indirgeme sırasında Berk Sistem'e yeni bir istemci dahil olamaz. Diğer bir deyişle istemcilerin sisteme dahil olması ve terketmesi şu an için yürütme zamanında mümkün değildir. Bunun en büyük sebebi performanstır. Bu katı yapının daha sonraki geliştirme çalışmalarında daha esnek bir hale getirilmesi faydalı olacaktır.

4.4. Berk Sistem'in Olgu Yapısı

Tablo 3.1'de verilen söz dizimi kuralları yürütme için pek uygun değildir. Bundan dolayı bütün cümleler istinasız olarak özel yapılı olgulara (facts) çevrilirler. Bu çevirme işlemi derleme işleminden hemen önce bir önişlemci sayesinde gerçekleştirilir. Bu olguları bundan sonra *berk_fact* olguları olarak adlandıracağız.

berk_fact olgusunun genel yapısı son derece sadedir ve aşağıda verilmiştir.

```
berk_fact(Head, Guard, BodyList).
```

Cümledeki baş (head) yapısı, hiç değiştirilmeden, olduğu gibi birinci argüman olarak olgu içine yerleşir. Fakat bütün @/2 operatörleri baş üzerinde dağıtılır.

Guard yapısında da hiçbir değişiklik yapılmaz. Olduğu gibi ikinci argümana yerleşir. Eğer cümlede *guard* mevcut değilse *true* varsayılır.

Ana değişim gövde çağrıları üzerinde yapılır. Her bir gövde çağrısı önce @/2 üzerinde dağıtılır. Sonra *condition* üzerinde yer alan her ifade

```
call(NeedAnd, NeedOr, WaitAnd, WaitOr, Goal, Share, u1)
```

şeklindeki *call/7* yapısı içine yerleştirilir. Bu yapı derlenecek olan en son yapıdır. Örnek olarak, *main/0* cümlesi için *berk_fact/3* yapısının nasıl kurulduğu aşağıda verilmiştir.

```
main@[fulcrum, phantom] ←
                    instr(X, g)@fulcrum with share(X),
                    instr(Y, g)@phantom with share(Y),
                    merge(X, Y, Z)@mig with [ need_or([X, Y]), share(Z) ],
                    outstr(Z)@tornado with need(Z).

dummy@mig.
dummy@tornado.
```

Yukarıdaki örnekte *fulcrum*'dan X ve *phantom*'dan da Y akışları (streams) yardımı ile bilgi akmaktadır. *mig* adlı istemcide akan bu bilgiler birleştirilmekte ve sonuç akış *tornado* makinesine gönderilmektedir. *tornado* makinesi ise içinden bilgi akan bu stream'leri ekrana yazmaktadır. Örnekteki her bir yüklem ileriki bölümlerde ayrıntılı olarak açıklanacaktır. Bu örnek için şimdilik sadece önişlem ile ilgileneceğiz.

Önişlem programı yine Prolog ile yazılmış olup, tüm sistemden bağımsız çalışan bir programdır. Aşağıdaki komutla,

```
$ berk_preproc  ConcPrologFileName  ServerIPorName  ServerPort
```

ConcPrologFileName adlı dosya için önişlem yapılır. Biz örneğimizi *berk_exam.pl* dosyası içine yazdık. İkinci argüman ise sistemin çalışacağı ağdaki sunucunun adıdır. Bu test için *localhost* adlı makine, yani kendi sistemimiz sunucu olarak kullanılmıştır. Son argüman ise hizmet verilecek olan portun numarasıdır. Örneğimiz aşağıdaki komut ile önişleme tabii tutulmuş ve 5 adet geçici dosya (temporary file) üretilmiştir:

```
$ berk_preproc berk_exam.pl localhost 1234
```

Önişlem iki ayrı geçiş (pass) ile yapılır. İlk geçişte *Head* önünde bulunan *@/2* operatörleri incelenerek sistemde kaç adet istemci olduğu tespit edilir. Yukarıdaki örnekte *main/0* önünde sadece iki adet istemci adı mevcuttur. Fakat aslında gövde incelenirse 4 adet istemci olduğu kolayca görülebilir. Önişlemciye iki adet daha istemci olduğunu belirtmek için dummy@mig. ve dummy@tornado. gibi iki ayrı olguyu sisteme tanıtıyoruz.

Böylece önişlemci, baş analizi yaparken diğer iki istemci adını da tespit etmiş olur. Aslında bu pek nadir karşılaşılacak bir durumdur. *dummy/0*, adından da anlaşılacağı gibi hiçbir etkisi bulunmayan bir olgudur. Yukarıdaki iki *dummy/0* olgusu aşağıdaki gibi 3 farklı biçimde de yazılabilir.

```
dummy@[mig, tornado].

true@mig.
true@tornado.

true@[mig, tornado].
```

Buradan da anlaşılacağı üzere *dummy/0* ile *true/0* olguları aynı semantiğe sahiptir.

Aslında gövde içindeki çağrılar incelenerek de bütün istemciler tespit edilebilir. Fakat ne baş ne de gövde çağrılarında istemci adı geçmeyebilir veya sistemde daha sonra kullanılmak üzere boşta bekleyen bir makine olabilir. Daha da önemlisi p@X gibi bir çağrıda X'in hangi istemci olacağı ancak yürütme zamanında tespit edilebilir. Bütün bu problemlerden kurtulmak için *dummy/0* tanımı geliştirilmiştir.

Ön işlemci tespit ettiği her istemci adı için *<İstemciAdı>_berk.pl* adında geçici dosyalar oluşturur. Ayrıca server_tmp.pl adlı başka bir geçici dosya daha kurulur.

Örneğimiz için elde edilen dosyaları sıra ile inceleyelim. *server_tmp.pl* dosyasının yapısı aşağıdaki gibidir.

```
% server_tmp.pl
my_no_name(1, mig).
my_no_name(2, tornado).
my_no_name(3, fulcrum).
my_no_name(4, phantom).
```

Her istemciye bir adet numara verilir. Bu numaralar sunucu tarafından bilinmek zorundadır. Bu dosya doğrudan sunucu üzerinde çalışacak olan programa eklenecektir. *my_no_name/2* yapıları her istemciye verilen numara ve istemci adından oluşur.

İstemciler her zaman 1'den başlayarak numaralandırılır. Sunucunun numarası 0'dır. Bu numaralar daha sonra değişkenleri dondururken (freezing) kullanılacaktır.

Oluşturulan diğer dört geçici dosyanın adı, biraz önce bahsettiğimiz uylaşım gereği

tornado_berk.pl, phantom_berk.pl, mig_berk.pl, fulcrum_berk.pl

olacaktır. Her bir dosyanın iç yapısı hemen hemen aynıdır. main/0 cümlesi *fulcrum* ve *phantom*'da bulunacaktır. Önce fulcrum_berk.pl dosyasını inceleyelim. Bu dosyanın yapısı tablo 4.1'deki gibi olacaktır.

Tablo 4.1'de okumayı kolaylaştırmak için, Prolog tarafından *_Number* şeklinde verilen değişken adları el ile değiştirilmiştir. Ayrıca satır numaraları verilmiş olup, satırlar derli toplu bir biçimde düzenlemiştir. Tabii ki bu düzenlemelerin genel çözüme hiçbir etkisinin olmadığı aşikardır.

2, 3 ve 4. satırlarda gerekli olan operatörler tanıtılmıştır. 5 ve 6. satırlarda olduğu gibi her bir istemcinin bir adet *numarası* ve *adı* mevcuttur. Daha önce bahsedildiği gibi numara ve adlar bütün sistem boyunca sadece bir tanedir, çakışma mevcut değildir.

Tablo 4.1: Önişlemciden geçmiş bir program örneği

```
1 % fulcrum_berk.pl

2 :- op(1150, xfx, '|').
3 :- op( 500, xfx, 'with').
4 :- op( 999, xfx, '@').

5 my_name(fulcrum).
6 my_no(3).

7 berk_fact(main,
8       true,
9       [call([], [],    [], [], instr(X,g),   [X], fulcrum ),
10       call([], [],    [], [], instr(Y,g),   [Y], phantom ),
11       call([], [X,Y], [], [], merge(X,Y,Z), [Z], mig     ),
12       call([Z],[],    [], [], outstr(Z),    [],  tornado)
13      ]
14      ).

15 berk_fact(call(Body), true, NewBody) :-
16 berk_comma_to_list(Body, BodyList),
17 berk_distribute_body(BodyList, [], Body1),
18 berk_make_call(Body1, NewBody).

19 berk_start :-
20    g_assign(var_no, 1),
21    berk_init_client('localhost', 1234, fulcrum),
22    g_assign(mark, 2),
23    berk_que_loop([mark|QueTail], QueTail).

24 :- initialization(
25          (catch(berk_start, X, berk_end(X))-> true ;
26          berk_end('Premature Ended.'))
27          ).
```

15 ve 27. satırlar arası çözümü başlatmak için yardımcı kodlar içerirler. Her istemci, *initialization/1* direktifi ile başlar. Bu direktifin içine *catch/3* fonksiyonu eklenmiştir. Böylece sistemin kontrollü bir biçimde son bulması sağlanır. Buraya kadar anlatılan bütün satırlar, her istemcide ortaktır.

7 ve 14 satırları arasında *berk_fact/3* yapısı mevcuttur. İlk iki argüman her zaman baş ve gövde kısımlarından oluşur. Örnekte *guard* mevcut olmadığı için, yerine *true* yazılmıştır.

Gövde içindeki her bir çağrı

call(NeedAnd, NeedOr, WaitAnd, WaitOr, Goal, Share, Client)

şeklinde verilen *call/7* yapısı içinde saklanır. Açıkça görüleceği gibi *Goal* ve *Client* bilgileri kullanıcının program içinde yazmış olduğu *Goal@Client with Conditions* yapısı içinden elde edilir. *Conditions* içindeki yapılar da yukarıdaki gibi ayrıştırılarak *call/7* içine serpiştirilirler.

merge(X, Y, Z)@mig with [need_or([X, Y]), share(Z)] yapısı

call([], [X,Y], [], [], merge(X,Y,Z), [Z],mig)

yapısına çevrilir. *need_or([X, Y])* yapısındaki değişkenler listesi *call/7* yapısının ikinci argümanı yapılmıştır. *merge(X, Y, Z)* çağrısı ise 5. argüman olarak, *share(Z)*'deki değişken ise 6. argüman olarak atanmıştır. *call/7*'nin son argümanı ise bu çağrının hangi istemcide indirgeneceğini tespit eder. Buna göre

call([], [X,Y], [], [], merge(X,Y,Z), [Z], mig)

ifadesi aşağıdaki gibi yorumlanacaktır:

. 7. argüman olan *mig* değeri kullanılarak *merge(X, Y, Z)* ifadesi indirgenmek amacı ile *mig* adlı istemciye gönderilecektir.

. *mig* adlı istemci bu çağrıyı alınca, 2. argümandan hareketle, X veya Y'nin kendisine ulaşmasını bekler.

. *mig* indirgemeyi tamamladıktan sonra, 6. argümandan hareketle, elde etmiş olduğu Z değerini diğer istemcilerle paylaşmak amacı ile sunucuya gönderecektir.

call/7'nin ilk dört argümanı indirgenecek olan çağrının değişkenleri tarafından sağlanması gereken koşulları tespit eder. Diğer bir deyişle *merge(X, Y, Z)* indirgenmeden önce X veya Y'den en az birinin kendisine ulaşmasını bekler. *call/7*'deki altıncı argüman ise indirgeme tamamlandıktan sonra sunucuya gönderilmesi gereken değişkenleri belirtir. Diğer bir değişle *merge/3*'ün solundaki ilk dört argüman indergenmeden önce, sonraki argüman ise indirgenmeden sonra işlenecek argümandır.

Örnekteki *call([Z],[], [], [], outstr(Z), [], tornado)* yapısında ise *outstr(Z)* indirgenmeden evvel Z değişkeninin kendisine gelmesini bekleyecektir. Z değişkeni sunucudan kendine gönderildikten sonra hemen uyanacak, gelen bilgiyi ekrana yazacak ve akıştan başka bilgi gelene kadar kendini askıya alacaktır. *outstr(Z)* indirgendikten sonra hiçbir değişkeni paylaşmayacağı için *call/7*'nin altıncı argümanı [] (boş liste) olarak belirtilmiştir. Diğer bir deyişle *outstr/1* çağrısı indirgendikten sonra hiçbir paylaşılan değişken meydana çıkmayacaktır, hiçbir değişken üretimi de yapılmayacaktır.

Örneğimizdeki *main/0* yüklemi, *main@[fulcrum, phantom]* başlığına sahiptir. Bu demektir ki bu cümle hem *fulcrum* hem de *phantom* istemcilerinde yüklü olmalıdır. Şu ana kadar *fulcrum_berk.pl* adlı geçici dosya anlatıldı. *main/0* yüklemi aynı zamanda *phantom* adlı istemcide de bulunacağı için bütün bu olguların tamamı *phantom_berk.pl* dosyasında da bulunacaktır. Böylece *main/0*'ın *phantom* adlı istemcide bulunacağı garanti altına alınacaktır.

mig ve *tornado* istemci adları, ilk ve son defa *dummy/0* olgusu ile kullanılmıştı. *mig_berk.pl* ve *tornado_berk.pl* dosyaları, 7-14 satırları arası hariç benzer olacaktır. Tabii ki *my_name/1* ve *my_no/1* olgularının argümanları farklı olacaktır. 7-14 satırları arasında ise *berk_fact(dummy,true,[]).* yapısı bulunacaktır.

Herhangi bir cümlenin gövdesinde, *main@tornado* gibi bir çağrı yaparsak bütün sistem hata ile bitecektir. Çünkü *tornado* içinde *main/0* yüklemi mevcut olmayacaktır.

4.5. Diğer Yardımcı Olgular

Bütün istemci kodlarında aşağıdaki gibi bir yapı bulunur.

```
19 berk_start :-
20   g_assign(var_no, 1),
21   berk_init_client('localhost', 1234, fulcrum),
22   g_assign(mark, 2),
23   berk_que_loop([mark|QueTail], QueTail).
```

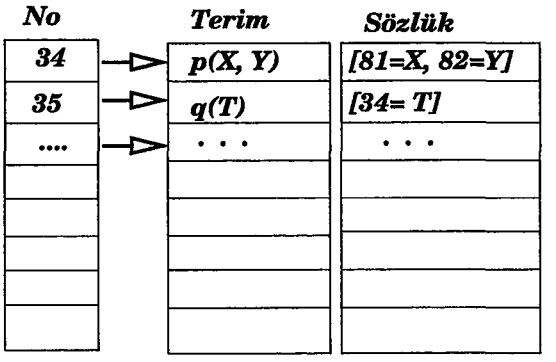
Bu yapıları kısaca inceleyelim. Bir istemci çözüme başlamadan önce *berk_start/0* yüklemi ile başlangıç işlemlerini yapar. İlk önce *var_no* global değişkeni 1 olarak atanır. *my_no/1* ve *var_no* global değişkeni ortak kullanılarak bütün değişkenlerin farklı bir numara ile numaralanması sağlanır. Bu sayede herhangi bir istemcide üretilen bir değişken her zaman bütün sistem içinde tek bir numara ile temsil edilir. Her değişken üretiminden sonra *var_no* değeri 1 artırılır. Değişkene her zaman

$< client\_no, local\_variable\_no >$

formatı ile oluşturulan bir numara verilecektir. Bu konu geniş olarak *freezing* başlığı altında anlatılacaktır.

berk_init_client('localhost', 1234, fulcrum) çağrısı ile sunucuya bağlantı talebinde bulunulur. Bağlantıda bulunulacak sunucu adı *localhost* olarak verilmiştir. Sunucunun servis vereceği port olarak da *1234* seçilmiştir.

Aşağıda tanım 1'de çözüm ağı tanımı verilmiştir. Aynı çözüm ağı içindeki bütün istemcilerin aynı sunucu adını ve aynı portu kullanmaları zorunludur. Sunucu adı aynı olmasına rağmen farklı bir port numarası verilirse, bu ayrı bir çözüm



Şekil 4.4: Sunucu tarafında değişkenlerin saklanması

ağına karşı gelir. Diğer bir deyişle bir sunucu aynı anda birden fazla çözüm ağına destek verebilir.

Tanım 1 (Çözüm ağı) *Bir ve yalnız bir sunucu, en az bir adet istemci ve bir port numarasından oluşan yapıya "çözüm ağı" denir.*

Her bağlantı isteğinden sonra, istemci kendi adını da göndermek zorundadır. *fulcrum* argümanı ile istemci kendi adını sunucuya gönderir. Bu adımdan sonra, artık bu istemci çözüm ağının bir parçası olacaktır.

Dikkat edilirse hiçbir adımda hata denetimi yapılmamıştır. Örneğin hatalı port numarası veya sunucu adından dolayı bağlantı gerçeklenmeyebilir. Bu gibi hata durumları her zaman *catch/throw* ifadeleri ile tespit edilir ve gerekli işlemler icra edilir. Şu tür çağrılarla da çözüm başlatılır:

```
g_assign(mark, 2), berk_que_loop([mark|QueTail], QueTail).
```

"mark global" değişkeni ana çözüm döngüsü başlığı altında anlatılacaktır. Kısaca değinmek gerekirse, *mark* değeri *2* değerini alırsa, port'lara bilgi gelene kadar, istemci kendini askıya alacaktır.

Önişlemci programı, diğer programlarla birlikte ekteki diskette verilmiştir.

4.6. Değişken Numaralandırma Yöntemi

Bir çözüm ağı içinde geçen herhangi bir değişken aynı anda iki veya daha fazla istemci tarafından kullanılabilir. Örneğin,

```
p ← q(X)@fulcrum,
    r(X)@phantom.
```

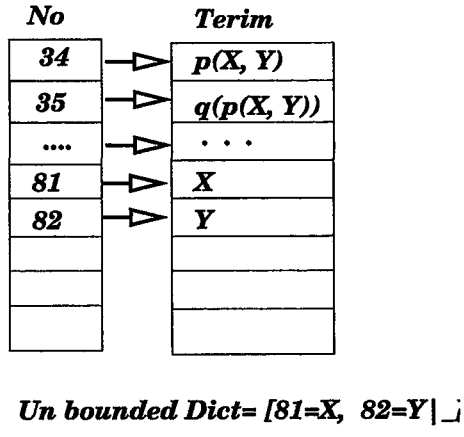
yükleminde X değişkeni iki ayrı istemcide kullanılmaktadır. Fakat değişkenler kendi *global stack (heap)* alanında anlamlıdır. Bir değişken bu alandan çıktığı an anlamını kaybeder. Değişken ismi ile kendisi arasında hiçbir ilişki mevcut değildir. Bir değişkenin birden fazla istemcide temsil edilebilmesi için, bu değişkene bir tamsayı atanır. Bu tamsayı bütün çözüm ağı boyunca tektir. Bu özellik kural 1'de dile getirilmiştir.

Kural 1 (Numaralandırma) *Çözüm ağı içinde farklı istemcilerde bulunan aynı sayılar aynı değişkeni temsil ederler.*

Sunucuyu *ağ alanı (network heap)* gibi düşünebiliriz. İhtiyacı olan istemciye, derhal istenen değişken gönderilir. Bu özellik çözüm ağının ve Berk Sistem'in temel taşlarından birini oluşturmaktadır. Bu özellik kural 2'de tespit edilmiştir.

Kural 2 (network heap) *Çözüm ağı içinde bulunan sunucu, bütün* paylaşılan değişkenlerin (shared variables) *birer kopyasını kendi içinde barındırır.*

Şekil 4.4'de sunucu tarafında değişkenlerin nasıl saklandığı sembolik olarak verilmiştir. Her terimin mutlaka bir adet kendine mahsus, özel (unique) numarası mevcuttur. Aynı zamanda her terime bir adet sözlük iliştirilmiştir. Terim içinde bulunan bütün *unbounded* değişkenlerin numaraları bu sözlük içinde mevcuttur. Şekilde, T değişkenine *34* numarası verilmiştir. Aynı zamanda *34* numaralı değişken ise p(X, Y) terimine karşı gelir.



Şekil 4.5: İstemci tarafında değişkenlerin saklanması

Bütün değişkenler sadece istemcilerde üretilirler. Bundan dolayı, değişkenlere özel bir numaranın verilmesi istemcinin sorumluluğu altındadır. Üretilen her paylaşılan değişkene, istemci tarafında hemen bir numara verilir.

Şekil 4.5'de temsil edildiği gibi, X için *81* ve Y için *82* sayıları üretilmiş olsun. *p(X, Y)* terimine karşı gelen *34* numaralı değişken, içerdiği yapı ve yapı içindeki değişkenlerin adları ve numaraları sunucuya gönderilir.

p(X, Y) terimi istemcinin kendi global yığın alanında da bulunmakta ve standart Prolog tarafından kullanılmaktadır. Bütün bu anlatılanları temsili olarak

variable(34) = p(X, Y) / [X=81, Y=82]

şeklinde gösterebiliriz. Liste içindeki ifadeyi *sözlük (dictionary)* olarak adlandıracağız. Bir paylaşılan değişken ve içeriği bu sözlük olmadan hiçbir anlam ifade etmez. Bundan dolayı istemciler arası terim alışverişinde, terim ile birlikte mutlaka terime ait olan sözlük de beraber taşınır. Bu özellik kural 3'de ifade edilmiştir.

Kural 3 (Değişken Taşıma) *Çözüm ağı içinde taşınan bir değişkene her zaman bir sözlük eşlik eder.*

Şekil 4.4'e tekrar geri dönersek, bu şekilde, sunucu tarafında p(X, Y) değerinin nasıl temsil edildiği ve saklandığı gösterilmiştir. Okuma kolaylığı olsun diye değişken adları yine, istemcide olduğu gibi, X ve Y olarak yazılmıştır. Aslında sunucu tarafında bu değişkenler artık bambaşka adresleri, konumları ifade ederler.

Sunucu her zaman global bir alanda bütün değişkenlerin numarasını ve yığın (heap) adresini saklar. Böylece son derece hızlı bir biçimde istenilen bir değişkene ve değerine ulaşılabilir.

Şekil 4.5'de de gösterildiği gibi, sunucudan gelen terim olan p(X, Y) değeri bilinen klasik yığında saklanır. Fakat buradaki her değişken aynı zamanda global alandaki bir sayıya işaret eder. Global alandaki her sayı ise yığına bir gösterge ile bağlıdır. Böylece sayı belli iken değişkene ulaşmak son derece hızlı ve kolay olur.

Sunucu tarafında değişkenler, numaraları, içerikleri ve sözlük listesi ile beraber saklanır. Fakat istemci tarafında sözlüğün ayrıca saklanmasına gerek yoktur. Şekil 4.5'de de görüleceği üzere, değişkenlerin saklandığı tabloda ayrıca bir sözlük sütunu mevcut değildir. Ek olarak sunucudan farklı olarak *unbounded* değişkenlerin bir listesi ayrı bir alanda saklanır.

Özetleyecek olursak istemciler arasında dolaşan bütün terimler mutlaka bir adet sözlük ile beraber dolaşmak zorundadırlar. Eğer terim içinde *unbounded* değişken yok ise, sözlük boş kümeden oluşacaktır.

Bir istemci, bir cümleyi indirirken her türlü ortak değişkene özel bir numara vermekle mükelleftir. Kural 4'de değişkenlere numara verme kuralı verilmiştir.

Kural 4 (Değişken numaralandırma) *Çözüm ağı içindeki her bir değişken <8 bit istemci numarası><24 bit değişken numarası> şeklinde tanzim edilmiş 32 bitlik bir tam sayı ile temsil edilir.*

Bu kuralı biraz daha açalım. Her istemciye birden başlamak üzere bir numara verilir. Bu numaraya *my_no* diyelim. Daha sonra her istemci içinde 1'den

itibaren saymaya başlayan bir başka numara daha olsun. Buna da *var_no* diyelim. <8 bit my_no> <24 bit var_no> şeklinde tanzim edilmiş 32 bitlik bir tamsayı üretilir. Üretilen her değişken için *var_no* değeri bir artırılır. *my_no* her zaman sabittir ve her bir istemci için önişleme sırasında tespit edilir.

Bu yöntemle her istemci, kendi başına, bağımsız olarak ürettiği her değişkeni numaralandırır. Fakat bu yöntemin iki büyük sakıncası vardır. Birincisi çözüm ağına en fazla *255* adet istemci katılabilir. İkincisi ise bir istemci en fazla $2^{24} - 1$ adet değişken üretebilir.

4.7. Dondurma (Freezing)

Bir terimi *ground* hale getirmek için bu terimin bütün değişkenleri *ground* bir terim ile değiştirilir. Bu işleme *dondurma (freezing)* denir.

Bu işlemin tersine de *eritme (melting)* denir.

Ertime işlemini kolaylaştırmak için, *dondurma* sırasında her değişkene karşı gelen bir sayı tutulur. Böylece eritme işlemi sırasında aynı değişkenlere aynı adresin verilmesi sağlanmış olur.

Bir terim içindeki bir *unbounded* değişken kendi heap alanı içinde anlamlıdır. Heap dışına çıkıldığı an bu değişkenin hiçbir anlamı kalmaz. Bundan dolayı bu değişkenlerin *ground* terim haline getirilmesi gereklidir. Fakat geri dönüşümünün de olması gerektiği açıktır.

Genelde dondurma işlemi için değişkenlere *$VAR(i)* yapısı atanır. Standart Prolog, ayrılmış kelimeleri *$* karakteri ile başlatır. Bundan dolayı *$VAR(i)* yapılarına da *ayrılmış kelime (reseverd word)* gözüyle bakılabilir.

p(X, q(X, Y)) terimini dondurmaya çalışalım. Bunun için tamamen keyfi olarak X yerine *$VAR(1)* ve Y yerine de *$VAR(2)* yazalım. Bu durumda örnek terim

```
p($VAR(1), q($VAR(1), $VAR(2))
```

şekline bürünecektir.

Bu terimin *ground* olduğu açıktır, hiçbir *unbounded* değişkene sahip değildir. Artık bu terim başka bir ortama kolaylıkla taşınabilir. Taşınan yerde ise eritilir ve p(K, q(K, H)) gibi bir terim elde edilir. Değişken adları farklı olmasına rağmen iki terim tamamen denktir.

Dondurma işleminin en büyük faydalarından birisi *nonground* değişkenlere sahip terimlerin başka ortamlara taşınabilmesidir. Taşınan yerde eritme işlemi yapılarak aynı terimin anlamlı bir biçimde elde edilmesi sağlanır.

Berk Sistem ise terimleri istemciler arasında taşımak için benzer bir teknik kullanır. Ek olarak, bir istemcide bulunan bir değişkenin başka bir istemcide de aynı değişken olmasının garantilenmesi gerekir. Berk Sistem bunu sağlamak için bütün değişkenlere özel bir numara verir. Aynı zamanda bir terimi hiçbir zaman gerçek anlamda dondurmaz. Çünkü donan bir terim başka bir makineye gönderilmesine rağmen, halen kendi istemcisi üzerinde kullanım halindedir. Değişken donduğu an istemci içinde kullanılamaz bir hale gelir.

```
p@fulcrum ← q(X)@phantom,
            r(X)@fulcrum,
            ...
```

Yukarıdaki örnekte *p/0* cümlesinin *fulcrum* adlı makinede yürütüleceğini kabul edelim. q(X) çağrısı indirgenmek amacı ile *phantom* adlı istemciye gönderilecektir. Bu çağrının terimi gönderilmeden önce dondurulur. Klasik dondurma işlemi uygulanırsa, *q($VAR(1))* gibi bir *unbounded* terim elde edilir. Bu terim problemsiz bir biçimde TCP/IP üzerinden *phantom* adlı makineye, indirgenmek üzere gönderilir. Fakat *fulcrum* makinesinde bulunan r(X) çağrısının terimi de, otomatik olarak *r($VAR(1))* haline gelecektir. Eğer *r($VAR(1))* terimi başka bir istemciye gönderilecekse bir problem yoktur. Hazır olarak donmuş olan bu terim hemen gönderilir. Fakat, eğer bu terim *local* makinede indirgenecek olursa

r($VAR(1)) teriminin tekrar eritilmesi gerekir. Bu da büyük zaman kaybına sebep olur.

Bu tür bir problem pek çok yol ile engellenebilir. İlk akla gelen çözümlerden biri, donacak olan terimin kopyasının alınması ve kopyasının dondurulmasıdır. Fakat burada da bir zaman kaybı meydana gelecektir.

Berk Sistem hem *dondurma* hem de *eritme* işlemini tek bir çatı altında toplamıştır. Üstelik *dondurma* işlemi gibi değişkenleri *ground* hale getirmez, dolayısı ile bu işlemin istenmeyen yan etkisi yoktur. Sonuç olarak *eritme* işleminde elde edildiği gibi bir indeks listesi elde edilir.

Örneğin, p(Y, p(T), Z, X, T, a(X)) terimini önerdiğimiz gibi dondurursak [1= X, 12= T, 13= Z, 2= Y] listesini elde ederiz. Orjinal terime ve değişkenlerine hiç dokunulmamıştır. Her değişkene istemci tarafında bir adet numara verilmiştir. Bu numara daha önce bahsedilen ve bütün çözüm ağı boyunca geçerli olan özel bir sayıdır. *Dondurma* işlemi bitince *Numara=Değişken* elemanlarından oluşan bir liste elde edilir. Diğer istemciye bu terim gönderilirken bu liste ile beraber gönderilir. Bu numara, bütün çözüm ağında aynı değişkeni temsil etmek için kullanılacaktır.

Aslında bu işlem bir *dondurma* işlemi değildir. Çünkü işlem bittiğinde, orijinal terim içinde hala *unbounded* değişken mevcut olacaktır. Ayrıca bu işlem bir eritme işlemi de değildir, çünkü orijinal terim bir *ground* terim değildir.

Tekrar p(Y, p(T), Z, X, T, a(X)) terimini ve [1= X, 12= T, 13= Z, 2= Y] listesini gözönüne alalım. Bu listeyi sözlük olarak adlandırmıştık. Sözlük içindeki her eleman *No=Var* olarak temsil edilmiştir. Sözlük içindeki her değişkene *$VAR(No)* ataması yaparsak orijinal terimi klasik anlamda dondurmuş oluruz. Örneğin, X= $VAR(1), T= $VAR(12), Z= $VAR(13) ve Y= $VAR(2) ataması yaparsak p($VAR(2), p($VAR(12), $VAR(1), $VAR(12), a($VAR(1))) *ground* terimini elde ederiz. Bu da klasik *dondurma* işlemi ile elde edilecek terimin aynısıdır.

Tablo 4.2: Dondurma işlemi

```
% ?- berk_freeze(+Global, +Term, -Local).

berk_freeze(Global, Term, Local) :-
   compound(Term),
   !,
   functor(Term, _, N),
   berk_freeze(N, Global, Term, Local).

berk_freeze(Global, X, Local) :-
   var(X),
   !,
   berk_get_var_number(Global, X, N),
   berk_add_no_var_to_dict(Local, N=X).

berk_freeze(_, _, _).

%-----------------------------

berk_freeze(0, _, _, _) :-
   !.

berk_freeze(N, Global, Term, Local) :-
   arg(N, Term, Arg),
   berk_freeze(Global, Arg, Local),
   N1 is dec(N),
   !,
   berk_freeze(N1, Global, Term, Local).
```

Ayırca bu ground terimi eritirsek p(Y, p(T), Z, X, T, a(X)) orijinal terimini[1] ve $VAR(1)= X, $VAR(12)= T, $VAR(13)= Z, $VAR(2)= Y] sözlüğünü elde ederiz. Bu sözlük içinden *$VAR/1* yapısını atarsak önerdiğimiz dondurma (freeze) yöntemi ile elde edilmiş olan orijinal terim ve sözlüğe dönmüş oluruz. Önerilen dondurma yöntemi tablo 4.2'de verilmiştir.

Dondurma işlemi yapılırken, daha önceden dondurulmuş olan değişken numaraları da gerekli olacaktır. Birinci argüman olarak,

[1]Okuma kolaylığı olsun diye aynı değişken isimleri kullanılmıştır

Global= [1=X, 2=Y, 3=P | _]

örneğindeki gibi, daha önce numaralanmış olan değişkenlerin listesi gelir. Daha önceki *dondurma* işleminde X, Y ve P değişkenlerine sıra ile *1*, *2* ve *3* sayıları iliştirilmiştir. İşlem kolaylığından ve esnekliğinden dolayı *tamamlanmamış liste (uncompleted list)* kullanılmıştır.

Dondurulacak terim ikinci argüman ile gelir. Örneğimizde terim,

Term= p(Y, p(T), Z, X, T, a(X))

olsun. Dondurma işlemi bittiğinde, üçüncü argüman olarak *Dict* listesi elde edilir. Örnek terim için

Local= [1= X, 12= T, 13= Z, 2= Y | _]

listesi elde edilir. Dikkat edilirse X ve Y için yeni değişken numarası talebinde bulunulmamıştır. Gerekli numaralar daha önce tespit edilen numaralar içinden arama yapılarak elde edilmiştir. Fakat T ve Z değişkenleri ilk defa dondurma işlemine tabii tutulacaktır. Bundan dolayı iki yeni numara talep edilmiştir.

Sistemdeki temel yüklemlerden birisi *berk_get_var_number(+Global, +X, -N)* yüklemidir. X değişkeni önce *Global* listesi içinde aranır eğer varsa, bu değişkene karşı gelen sayı *N* ile geri döndürülür. Eğer X değişkeni *Global* listesi içinde mevcut değilse yeni bir numara talep edilir. İstemcinin verdiği bu numara hemen *Global* sözlüğünün sonuna *N=X* terimi olarak eklenir. Aynı zamanda bu numara *N* ile geri döndürülür. İlave olarak bu *N=X* değeri *berk_add_no_var_to_dict(+Local, +N=X)* yüklemi yardımı ile *Local* listesi içine eklenir.

Her yeni değişkene yeni bir numara verilir ve verilen bu numara Prolog'un global alanında saklanır. Örneğin 3 yeni değişkenimiz olsun. Bu değişkenlere verilen 3 yeni numara *varno= ['1', '2', '3']* şeklindeki tamamlanmış bir listede saklanır. Her yeni gelen numara liste başına eklenir.

Dondurma işlemi sırasında üretilen her yeni değişken *Global* listesi içine eklenir. İlk bakışta listenin ilerleyen çözüm süresi içinde çok büyük boyutlara ulaşacağı düşünülebilir. Fakat *Global* listesi içindeki her değişken *unbounded* ise bize gereklidir. *Bound* olmuş bir değişken hemen *Global* içinden silinir. Bu sayede *Global* listesinin boyu, çözülen probleme de bağlı olarak, genelde son derece kısadır.

4.8. Sunucudan Değişken Talep Edilmesi

Çözüm ağındaki en büyük problemlerden biri değişkenlerin istemciler arasında taşınmasıdır. Berk Sistem'de, istemciler ihtiyaç duydukları değişkeni her zaman sunucudan talep ederler. Bir istemci bir değişkeni başka bir istemciden talep edemez. Çünkü değişkeni talep eden istemci, değişkenin hangi istemci üzerinde hesaplandığını her zaman açık olarak bilemez. Bundan dolayı bütün değişkenler sunucudan talep edilir. Buradan son derece basit bir sonuç çıkmaktadır: Bütün paylaşılan değişkenlerin tam bir adet kopyası sunucu üzerinde bulunmalıdır. Böylece bir istemci, değişken ihtiyacını karşılamak için sadece sunucu ile muhatap olacaktır. Bu tekniğin uygulaması son derece basit ve etkili olmasına rağmen bazı sakıncaları mevcuttur.

En büyük problem bütün ortak değişkenlerin zamanla hızla artması ve sunucunun kaynaklarının tükenmesidir. Geliştirilecek bir *artık toplama (garbage collection)* tekniği bu problemin üstesinden gelebilecektir. Şu anda Berk Sistem'de *dağıtık artık toplama (distributed garbage collection)* mekanizması mevcut değildir.

Bu kadar büyük olmasa bile bir diğer sakınca da, bir değişkenin hedefine, yani talep edilen istemciye ulaşabilmesi için sunucudan geçmek zorunda olmasıdır. Bu da bir miktar zaman kaybına sebep olmaktadır.

Fakat paylaşılan değişkenlerin sunucu tarafında tutulmasının pek çok yararı da vardır. Aşağıda, bu faydalardan kısaca bahsedilmiştir.

Bir istemci, değişken talep edeceği zaman, değişkenin hangi istemcide hesaplandığını bilmesi gerekmez. Sadece sunucudan talepte bulunur.

Ortak değişkenler bütün çözüm ağı boyunca tek bir istemcide hesaplanır. Diğer istemciler ise sadece bunu kullanırlar ve tüketirler. Diğer bir deyişle bir tek istemcide üretilen değişken birden fazla istemcide tüketilebilir. Değişkeni üreten istemci, ürettiği değişkenin bir örneğini sunucuya gönderir. Sunucuya gitmeyen hiçbir değişken paylaşılan bir değişken değildir. Bu kısıt, kural 5 ifadesi ile verilmektedir.

Kural 5 (Paylaşılan değişken) *Bir değişkenin paylaşılan bir değişken olması için gerek ve yeter şart, bu değişkenin sunucuya gönderilmesidir. Bir değişken başka hiçbir çağrı tarafından kullanılmasa bile, void özelliğinde dahi olsa, sunucuya ulaştığı an paylaşılan değişken sıfatına bürünür.*

Kural 5'de verilen tanım, bilinen ortak değişken kavramını da değiştirmektedir. CCND dillerinde bir değişkenin ortak olması veya paylaşılması için aynı cümle içine iki farklı çağrıda geçmesi yeterlidir. Berk Sistem tarafında ise değişkenin sunucuya ulaşması yeterlidir. Sunucuya ulaşan her değişken, paylaşılan kabul edilmektedir.

Kimse talepte bulunmasa bile üretilen ortak değişkenler mutlaka sunucuya gönderilir. Bir değişken sadece bir kez üretileceği için, bu değişken sunucuya bir kez gelir ve gelen değişken sunucu tarafında saklanır. İstemcilerden talep oldukça ilgili değişkenler gönderilir.

Üretilen her ortak değişken sunucuya gelmek zorundadır. Eğer bir değişken birden fazla yerde üretiliyorsa çakışma analizi yapmak son derece kolaylaşır. Eğer birden fazla üretici varsa aynı değişken için üretilen değerler de aynı olmak zorundadır.

Sunucu çözümün genel gidişatını ve değişkenlerin almış olduğu değerleri kolaylıkla kontrol edebilir. Çünkü çözümde üretilen bütün ortak değişkenlere sahiptir.

Hata ayıklama (debug) işlemi sunucu üzerinden kolaylıkla yapılabilir.

Bir istemci talep edeceği değişkenin önce numarasını tespit eder. Değişken her zaman bu numara ile talep edilir. ?- berk_iste(+Number, +Alias) çağrısı ile talepte bulunulur. Değişkeni talep eden istemcinin adı *Alias* argümanı ile verilir.

Sunucudan talep edilen her bir değişken bir listede saklanır. Bu liste sayesinde aynı değişken, sunucudan sadece bir kez talep edilir. Eğer değişken daha önce bir kez talep edilmiş ise *?- berk_iste/2.* sorgusunun hiçbir etkisi olmayacaktır.

Bir değişken ilk defa isteniyorsa sunucuya need(No, Alias) iletisi gönderilir. *No* değeri de talep edilen değişkenler listesine eklenir. Değişken sunucudan geldiğinde bu *No* değeri hemen listeden atılır.

Değişken sunucudan talep edildikten hemen sonra, değişkeni bekleyen çağrı derhal *askı* durumuna geçer ve değişkenin gelmesini bekler. Örneğin, X değişkeninin numarası *1* olsun. Hemen need('1', fulcrum) iletisi sunucuya gönderilir. Daha sonra bu değişkenin talep edildiğini hatırlamak için *1* sayısı listeye eklenir. Örnekteki listeye göre daha önce *3* ve *5* numaralı değişkenler sunucudan talep edilmiştir. Değişken gelene kadar, bu değişkeni talep eden *p/1* çağrısı askı durumuna geçer ve değişkenin gelmesini bekler.

Şimdilik pek bir uygulama alanı bulunamamış olsa bile, bir istemci başka bir istemci için değişken talebinde bulunabilir.

Örneğin *fulcrum* adlı istemci ?- berk_iste('3', phantom). sorgusu ile *3* numaralı değişkeni talep edebilir. Fakat bu değişken *fulcrum* adlı istemciye değil, *phantom* adlı istemciye gönderilecektir.

Bu özel durumda bir değişkenin sunucudan gelebilmesi için *need/2* iletisi ile talep edilmesi gerekmez. Başka bir istemci, diğer bir istemciye, bir değişkenin gönderilmesini sağlayabilir. Bu durumda, değişkeni tüketecek olan istemci, değişken talep etmeden, değişkene sahip olacaktır. *need/1* iletisinin sunucuya

gitmesi gerekli olmadığı için son derece hızlı bir yöntemdir. Fakat bu tür değişkenlerin çözüm ağı içinde takip edilmesi ve programlanması çok daha zordur.

Örneğin, *fulcrum* adlı makinede olduğumuzu varsayalım.

```
main :- p(a(X)) with need(X).
```

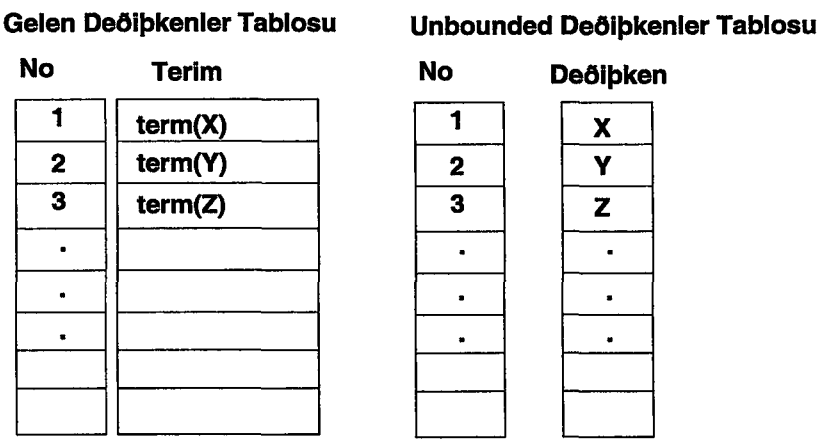
cümlesini gözönüne alalım. Eğer X değişkeni *fulcrum* içinde halen *unbounded* ise derhal sunucudan talep edilme işlemi yapılır. Eğer daha önce istenmemiş ise *need/2* mesajı sunucuya gönderilir ve *p/1* çağrısı askıya alınır.

Yine *fulcrum* makinesinde `main :- q(a(X)) with wait(X).` cümlesini ele alalım. Bu örnekte, *need/1* yerine *wait/1* yapısı konmuştur. Bu durumda sunucuya *need/2* iletisi gönderilmez. Değişkenin, sunucu tarafından *fulcrum*'a gönderilmesi beklenir. Sunucuya hiçbir talep mesajı gönderilmeden bekleme yapıldığı için, bu yöntemi *körleme bekleme* olarak adlandırabiliriz.

Şimdi ise *phantom* adlı makinede olduğumuzu varsayalım. `main :- r(X) with share [X with fulcrum].` cümlesini gözönüne alalım. `r(X)` çağrısı ile X değişkeni üretilecektir. Üretilen bu değişken *[X with fulcrum]* deyimi sayesinde derhal sunucuya gönderilir. Fakat sunucu *with fulcrum* ifadesine bakarak, bu değişkenin *fulcrum*'a gönderilmesi gerektiğini anlar ve *fulcrum* tarafından *açık talep olmasa bile* bu değişkeni derhal *fulcrum*'a gönderir.

4.9. Sunucudan Gelen Değişkenin Değerlendirilmesi

Sunucu her türlü değişkeni `var(N, term(Term, Dict))` yapısı içinde istemcilere gönderir. *Term* adlı terim istemcide bulunan *N* nolu değişkene karşı gelmektedir. Taşınan bir terimin tek başına hiçbir anlamı yoktur. Bundan dolayı bilgisayarlar arasında taşınan her terime *Dict* adlı bir liste eşlik eder. *Term* içinde bulunan her bir değişkenin özel *(unique)* numarası mutlaka bu *Dict* listesi içinde mevcuttur. Bu *Dict* listesi olmadan *Term*'in hiçbir anlamı yoktur.

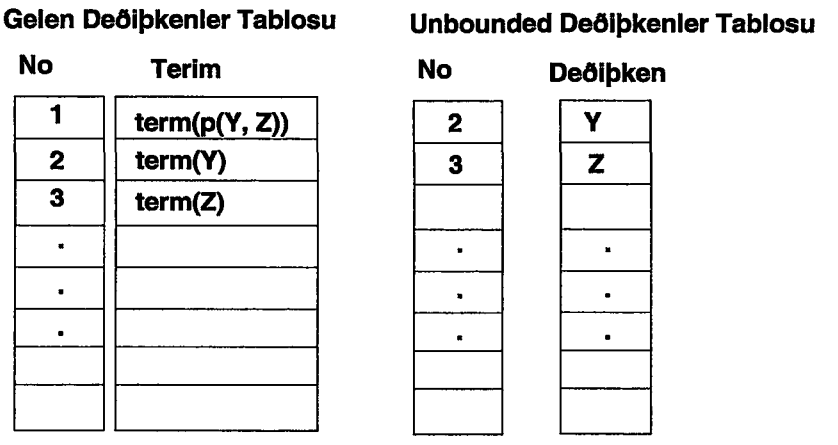


Şekil 4.6: İstemci tarafında değişkenlerin saklanması

var(1, term(p(Y, Z), [2= Y, 3= Z]) örneğini ele alalım. *1* numaralı değişken olarak p(X, Y) terimi gelmiştir. Buradaki Y ve Z değişkenleri ancak [2= Y, 3= Z] listesi ile anlam bulur. Önce *Dict* listesi içindeki her bir *No= Değişken* elemanı değişken tablosunda saklanır. Şekil 4.6'de örnek tablo verilmiştir. *2* ve *3* nolu satırlar doğrudan *Dict* içinde gelen elemanlardan kurulmuştur. 1 nolu satır ise aslında daha önce sunucudan talep edilen 1 nolu değişkeni temsil eder. Bu değişken p(Y, Z) olarak gelmiştir.

İstemci içindeki *unbounded* değişkenlerin bilinmesi son derece önemlidir. Bundan dolayı ayrı bir tabloda da bütün *unbounded* değişkenler saklanır. Sunucudan gelen *Dict* içindeki bütün değişkenler *unbounded* olduklarından, buradaki bütün elemanlar doğrudan unbounded değişkenler listesine eklenirler. Şekil 4.6'de *Unbounded Değişkenler Tablosu* gösterilmiştir.

Talep edilen değişken daha önce mutlaka *Gelen Değişkenler Tablosu*'nda *term/1* şeklinde saklıdır. Örnekte, *1* numaralı değişken term(X) yapısı içinde saklanır. Diğer bir deyişle, çözüm ağı içinde *1* numara ile temsil edilen değişken, aslında bu istemcide X değişkenidir. Sunucudan 1 numaralı değişken olarak p(Y, Z) geldiği için derhal X= p(Y, Z) ataması yapılır. X değişkenine p(Y, Z) atanmıştır. X değişkenine atama yapıldığı için artık bu değişkenin *unbounded* değişkenler listesi



Şekil 4.7: X değişkenine p(Y, Z) teriminin atanması

içinde bulunmasına gerek yoktur. X değişkeni derhal bu listeden atılır. Bu son durum şekil 4.7'de verilmiştir.

Unbounded Değişkenler Tablosu son derece kolay biçimde güncellenir ve uzunluğu çözüm süresi boyunca fazla artmaz. Fakat *Global Değişkenler Tablosu* çözüm boyunca sürekli olarak büyür. Bundan dolayı bu liste de GC'ye muhtaçtır.

Özetleyecek olursak, var(N, term(Term, Dict)) terimi sunucudan gelir. *Dict* içindeki her bir değişken *Global Değişkenler Tablosuna* ve *Unbounded Değişkenler Tablosu*'na eklenir. Ayrıca *Term* ifadesi de *Global Değişkenler Tablosu*'nda bulunan term(T) ifadesindeki T terimi ile *unify* edilir. İşte burada *çakışma* durumu varsa hemen ortaya çıkar. Diğer bir deyişle sunucu tarafında olduğu gibi istemci tarafında da çakışma denetimi kolaylıkla yapılabilmektedir.

4.10. Sunucuya Değişken Gönderilmesi

Çözüm ağı içindeki temel problemlerden birisi hesaplanan değişkenlerin sunucuya gönderilmesidir. Bir indirgeme işlemi başarı ile tamamlandığında ortak

değişkenler listesine bakılır. Bu listede yer alan atanmış değişkenler (bounded variables) sunucuya gönderilir.

Ortak değişkenler listesi programlama aşamasında her zaman kullanıcı tarafından açıkça belirtilir. main :- p(X, Y) with share [X, fulcrum-Y] örneğini ele alalım. İndirgeme aşamasından evvel, değişken numaralama sırasında [X, fulcrum-Y] listesi, [1= X, 2-fulcrum= Y] şeklindeki bir listeye çevrilir. Bir an için *p/2* çağrısının başarılı bir biçimde indirgendiğini ve X değişkeninin p(a(T)) ve Y değişkeninin de q(T) değerini aldığını varsayalım. Bu durumda ortak değişkenler listesi [1= p(a(T)), 2-fulcrum= q(T)] şeklini alacaktır.

Bu liste içinde bulunan ve değer almış olan her değişkenin hemen sunucuya gönderilmesi gereklidir. Fakat gönderilecek terimler içinde *unbounded* değişkenler mevcut olduğundan, doğrudan gönderme yapılamaz. Bundan dolayı gönderilecek terimlerin içinde bulunan bütün *unbounded* değişkenlere özel bir numara verilmelidir.

Örnekteki T değişkenine *3* sayısı atanmış olsun. Bu durumda p(a(T)) terimi [3= T] biçimindeki bir *Dict* listesi ile artık anlamlı hale gelir ve doğrudan sunucuya gönderilebilir.

Sunucuya gönderilen bir terim her zaman var(No, term(Term, Dict)) veya var(No-Alias, term(Term, Dict)) yapısı ile gönderilir. Örneğimiz için *1* ve *2* nolu değişkenler var(1, term(p(a(T)), [3= T])) ve var(2-fulcrum, term(q(T), [3= T])) şeklinde sunucuya gönderilir.

Her iki terim de sunucuda var(No, term(Term, Dict)) biçiminde saklanır. var(1, term(p(a(T)), [3= T])) şeklindeki terimler sunucudan talep edildikten sonra, *talep edene* gönderilir.

var(2-fulcrum, term(q(T), [3= T])) şeklindeki bir terim sunucu içinde var(2, term(q(T), [3= T]) formatında saklanır, fakat *fulcrum* bilgisinden dolayı hiçbir talep beklemeden ilgili yapı *fulcrum* adlı makineye gönderilir. Aynı zamanda daha sonra başka bir istemci bu değişkeni yeniden talep edebilir.

Buradaki en büyük problem numaralandırma işlemi sırasında daha önceden kullanılmış olan değişkenlerin numaralarının tespit edilmesidir. Eğer bütün değişkenler ilk defa kullanılıyor ise hiçbir problem yoktur. Fakat gönderilecek terim içinde geçen değişken eğer daha önce çözüm adımlarının içinde en az bir kez geçmiş ise büyük bir problem doğar. Daha önce kullanılmış olan bu değişkene verilen sayı mutlaka tespit edilmelidir. İşte bu problemin üstesinden gelebilmek için sistemde bulunan bütün *unbounded* değişkenlerin adı ve numarası tamamlanmamış bir listede saklanır. Böylece bir değişkene numara verilmeden önce bu liste taranır. Eğer değişken burada bulunamaz ise bu değişken yeni bir değişkendir ve ilk defa bu istemcide üretilecektir. Bu değişkene yeni bir numara verilir ve *unbounded* değişkenler listesine eklenir.

Unbounded değişkenler listesi içinde bulunan bounded değişkenler derhal listeden atılır. Bu sayede bu liste genelde son derece kısadır, zamanla büyümez.

Önerilen yöntemi aşağıdaki gibi özetleyebiliriz.

Ortak değişkenler listesi içinde bulunan No=Term veya No-Alias= Term şeklindeki ifadeler sunucuya gönderilecektir. *Term* mutlaka *bounded* olmalıdır. Fakat içinde *unbounded* değişken bulunabilir.

Term içindeki bütün *unbounded* değişkenler numaralandırılır. Numara verirken önce *unbounded* değişkenler listesi kullanılır. Burada bulunmayan her bir değişkene yeni bir numara verilir ve bu değişken ile yeni numara *unbounded* değişkenler listesine eklenir.

Her bir değişken var(No, term(Term, Dict)) veya var(No-Alias, term(Term, Dict)) yapısı ile sunucuya gönderilir.

Sunucu her iki yapıyı da var(No, term(Term, Dict)) şeklinde saklar. Ayrıca *Alias* içeren yapıyı ilgili istemciye talep olmadan gönderir.

4.11. Sunucuya Herhangi Bir Mesajın Gönderilmesi

Çözüm ağının performansını etkileyen en önemli faktörlerden birisi de mesaj gönderim tekniğidir. Bu tekniğin basit ve etkili olması gerekmektedir. Çözüm ağı içindeki sunucular berk_send_message(+Message) fonksiyonu ile sunucuya mesaj gönderirler. Bu fonksiyonun yüklemi aşağıda verilmiştir.

```
berk_send_message(Message) ← alias(server, _, Output, _),
                             writeq(Output, Message),
                             write(Output, '\n'),
                             flush_output(Output).
```

Message standart Prolog terimidir. Bu terim hiçbir işleme tabii tutulmaz. Mesela, gönderilecek bütün terim string'e çevrilmez, ya da değişkenler dondurulmaz.

Öte yandan, alias(server, Input, Output, HostName) olgusu, sunucu ile bağlantı kurulduğu an üretilir. İlk argümanda her zaman bağlantı kurulan makinenin sembolik adını verir. Bütün istemcilerde bu ad *server* sabiti şeklinde tespit edilmiştir. *Input* ve *Output* argümanları ise sunucu ile kurulan çift yönlü soket bağlantısının akış (stream) numaralarıdır. Örneğin,

Output= '$stream(4)', Input= '$stream(5)'

olabilir. Sunucuya gönderilecek her ileti *Output*, yani *'$stream(4)'* üzerinden gidecektir. Benzer şekilde sunucudan gelen her türlü bilgi ise *Input*'dan yani *'$stream(5)'* den okunacaktır. *HostName* değişkeninde ise sunucunun *'127.0.0.1'* gibi IP adresi veya *'localhost.localdomain'* gibi *domain* adı mevcuttur.

Şimdi, tekrar berk_send_message/1 cümlesine dönelim. alias(server, _, Output, _) çağrısı ile *Output stream* değeri elde edilir. Bu akış üzerinden sunucuya ileti gönderilecektir.

writeq(Output, Message) fonksiyonu ile bu *stream* üzerine bilgi yazılır. *write/2* yerine *writeq/2* arşiv fonksiyonu kullanılmıştır. Çünkü tırnak içinde yazılması gereken her şey mutlaka tırnak içinde gönderilmelidir. Aksi taktirde iletiyi alan taraftaki *read/2* fonksiyonu söz-dizim hatası verecektir.

Örneğin p('X') terimi sunucuya gönderilecek olsun. Eğer write(Output, p('X')) fonksiyonu kullanılmış olsaydı, sunucu bu terimi p('X') olarak değil p(X) olarak değerlendirecekti. Dolayısı ile istemci tarafında karakter katarı (string) olarak gözüken X değeri, sunucu tarafında değişken olarak değerlendirilecekti. *writeq/2* kullandığımız zaman 'X' ifadesi sunucuya tırnak işaretleriyle ile beraber gidecektir. *writeq/2* ile yazılan terim hemen sunucuya gönderilmez. *Çıkış Arabelleğinde (output buffer)* bekletilir.

Sunucu tarafı, bu bilgileri, Prolog'un klasik *read/2* fonksiyonu ile okuyacaktır. read/2 fonksiyonu, terim sonunda bir nokta ve bir de *white space* karakteri bekler. Bunu sağlamak için sunucuya gönderilen her terimin sonuna write(Output, '.\n') ile bir nokta ve bir "\n" karakteri eklenir. Bu iki karakter de ara-bellekte bekler. Hemen sunucuya gönderilmez.

flush_output(Output) fonksiyonu ile *Output* ara-belleğinde bekleyen bütün ifade sunucuya gönderilir.

Kullandığımız bu basit yöntemle herhangi bir terim hiçbir önişlemeye tabii tutulmadan diğer bilgisayara gönderilmektedir. *Output stream*'e yazılan ifadelerin, hemen gönderilmeden ara bellekte bekletilmesini set_stream_buffering(Output, block) fonksiyonu ile sağlıyoruz. Bu şekilde *Output stream* için blok bellekleme tekniğini kullanarak bilgilerin hemen gönderilmesini engelliyoruz. *flush_output/1* fonksiyonu ile de belleği boşaltıyoruz. Belleği boşaltmak demek içindeki bilgilerin bir kerede diğer bilgisayara gönderilmesi demektir.

Şu ana kadar istemcilerin sunucuya mesaj gönderme tekniği anlatılmıştır. Sunucu da tamamen aynı tekniği kullanarak istemcilere mesaj gönderir. Aradaki tek fark

alias(server, Input, Output, HostName) olgusunda birinci argüman yerine terimin gönderileceği istemcinin *alias* isminin bulunmasıdır.

4.12. Gelen Değişkenlerin Sunucu Tarafında Değerlendirilmesi

Bir istemci bir başka istemciye doğrudan değişken gönderemez. Bir değişken önce sunucuya gönderilir. Sunucu da gelen bu değişkeni diğer bir istemciye aktarır. Aynı zamanda sunucu kendisine gelen bütün değişkenleri saklar. Böylece çeşitli zaman aralıklarında talep edilen aynı değişken derhal ilgili istemcilere gönderilir. Bu yöntemde sunucu bütün ortak değişkenlere sahip olacağı için çözüm ağı için global yığın gibi davranır. Biz bunu ağ yığını (network heap) olarak adlandıracağız.

Ortak değişken sayısı arttıkça sunucuda biriken değişken sayısı da artacak ve sunucunun kaynaklarını hızla tükenecektir. Bu kaynak tüketimini engellemenin en iyi yollarından birisi atıkların toplanmasını (GC) uygulamaktır. GC uygulaması bu tezin kapsamı dışında tutulmuştur.

Sunucuya gelen bütün değişkenler var(No, term(Term, Dict)) veya var(No-Alias, term(Term, Dict)) yapısındadır. Her iki yapıda da ilgili terim Şekil 4.6'deki gibi *No* ve *$term(Term, Dict)* ikilileri halinde saklanırlar. Eğer *No* değişkeni daha önce gelmiş ise yeni ve eski değerler basitçe *unify* edilir. Böylece olabilecek çakışma durumları ortaya çıkarılabilir. Eğer bir değişken en az iki istemci tarafından üretilirse çakışma durumu ortaya çıkabilir.

var(No-Alias, term(Term, Dict)) yapısı özel bir öneme haizdir. Bu yapı yine yukarıda anlatıldığı gibi sunucu içinde saklanır. Aynı zamanda term(Term, Dict) yapısı *Alias* adlı istemciye gönderilir. *Alias* adındaki bu istemci bu değişkeni talep etmemiştir. Körleme bekleme yapmıştır.

4.13. Bütün Çözüm Ağının Dondurulması

Bütün çözüm ağında veri alışverişi ve hesaplanmanın tamamen ve geçici bir süre için durdurulması işlemine *çözüm ağının dondurulması* veya *hesaplanmanın dondurulması* diyeceğiz. Çözüm ağının dondurulması hem hata ayıklama (debug) amacı ile hem de *stop* ve *fail* gibi diğer *çözüm ağı kontrol işlemlerine* yardım amacı ile kullanılabilir.

Bütün çözüm ağı kontrol işlemleri (freeze, continue, stop, fail) sunucu tarafından denetlenir. Herhangi bir istemci veya konsol karşısındaki bir kullanıcı sunucudan *freeze* talebinde bulunabilir. Bu durumda sunucu derhal bütün istemcilere istisnasız *freeze* mesajı gönderir.

Bütün çözüm ağını dondurma işlemi istemci veya sunucu klavyelerinden *freeze* komutunu girerek yapılabilir. Ya da istemcilerin herhangi birinde çalışan bir program, sunucuya *freeze* komutunu göndererek çözüm ağını dondurma işlemini başlatabilir.

Kullanıcı, klavye ile herhangi bir ileti girebilir. Sunucuya bağlı olan klavye aynı zamanda çözüm ağına bağlı herhangi bir istemci gibi düşünülebilir. Böylece sunucu, kendi klavyesinden gelen *freeze* iletisini sanki herhangi bir istemciden gelmiş gibi kabul eder. Tabii ki klavyenin bir *alias* adı yoktur. Fakat sunucu ve bütün istemcilerde klavyeden herhangi bir ileti girilebilir. Bu da çözüm ağı ile kullanıcı arasındaki iletişimi çok kolaylaştırmaktadır.

Örneğin, çözüm ağı ayakta ve çözüm üretirken, kullanıcı klaveyeden *freeze* mesajı girebilir. Kısa bir süre sonra bütün çözüm ağı uykuya dalar. Bu durumda kullanıcı herhangi bir istemciden veya sunucudan klavyeyi kullanarak sistemin o anki durumunu (system state) inceleyebilir. Daha sonra da sunucudan *cont* komutunu girerek bütün sistemi ayaklandırır ve çözüm ağı kaldığı yerden çözüme devam edebilir.

Ya da bütün sistem uykuya daldıktan ve sistem durumu incelendikten sonra bütün çözümün *fail* olması istenebilir. Bu durumda istemci ve sunucu içindeki bütün çözüm bilgileri yok edilir ve çözüm ağı yeni bir çözüm için hazır halde bekler.

Tekrar *freeze* işlemine dönelim. Çözüm ağını dondurma işlemi klavyeden girilen veya istemci içinde gönderilen *freeze* komutu ile başlatılmış olsun. *freeze* iletisini alan sunucu derhal bütün istemcilere freeze mesajı gönderir. Artık bu andan itibaren freeze işlemi başlatılmış olur. Sunucu, donacak olan bütün istemcilerin adlarını bir listede saklar.

freeze iletisini alan istemci derhal sunucuya freeze_ok iletisi gönderir. Her freeze_ok iletisi gönderen istemci artık donmuş bir durumdadır. Bu istemci artık kendi içinde indirgeme yapmaz, sunucuya hiçbir zaman hiçbir amaçla ileti göndermez fakat gelen bütün iletilere açıktır.

Sunucunun elinde donması gereken istemcilerin listesi mevcuttur. Her freeze_ok mesajı gönderen istemcinin adı bu listeden düşülür. Bütün liste boşaldığında veya diğer bir deyişle donması gereken her bir istemci freeze_ok mesajı gönderdiğinde artık bütün çözüm ağı durmuştur. Bu durumda sunucu da kendini dondurmaktadır. İşte bu en son adımda bütün çözüm ağı donmuştur. Sunucunun konsoluna bütün çözüm ağının *donduğu* ikazı yazılır.

Sunucu bir istemciye *freeze* mesajı gönderdikten sonra, o istemciye artık herhangi bir mesaj göndermez.

Kullanılan bu yöntemde çözüm ağı içinde en önce istemciler donar, en sonda ise sunucu donar. *freeze* mesajını en önce alan istemci donan ilk istemcidir. TCP/IP protokolü altında veri alışverişi yapıldığı için *freeze* mesajı yoldaki diğer mesajların önüne geçemez. *freeze* mesajı, o anda yolda olan bütün mesajların en arkasından ilerler, hiç bir zaman önüne geçmez. UDP/IP protokolünde böyle bir problemin ortaya çıkma ihtimali mevcuttur. Ayrıca, sunucu freeze iletisini gönderikten sonra o istemciye bir daha mesaj göndermez. Tek istisna *continue* mesajıdır.

Aynı durum istemci tarafı için de geçerlidir. İstemci de *freeze* mesajını aldıktan sonra derhal *freeze_ok* mesajı gönderir ve sunucuya bir daha mesaj göndermez. Burada da, sunucuya mesaj gönderme konusundaki tek istisna sunucudan *continue* mesajının gelmesidir. Bu mesajdan sonra istemci tekrar uyanır ve kaldığı yerden çözüme devam eder.

Freeze işlemi çözüm ağı kontrol işlemlerinin her zaman ilk adımıdır. Bütün kontrol işlemlerinden evvel çözüm ağı uyutulur.

4.14. Donan Çözüm Ağının Uyandırılması

Bir çözüm ağı donduktan sonra prosesler arası mesaj alışverişi yapılamaz. Fakat sunucu klavyesi her durumda veri girişi kabul eder. Kullanıcı sunucu klavyesinden *continue* iletisi girerek bütün çözüm ağını tekrar ayağa kaldırır. Çözüm ağı, hesaplamaya kaldığı yerden devam eder.

Devam (continue) mesajını alan sunucu derhal bütün istemcilere *continue* mesajı gönderir ve kendini de uyandırır. *continue* mesajını alan istemciler hemen kendi içindeki hesaplama işlemlerine ve mesaj alışverişine kaldıkları yerden devam ederler.

4.15. Çözüm Ağının Temizlenmesi

Herhangi bir anda çözüm ağı içindeki bütün bilgilerin ve mesajların silinmesi istenebilir. Sistem bu hali ile açılıştaki durumuna gelir. Çözüm ağının temizlenmesi için sunucuya *clean* mesajı ulaşmalıdır. Bu durumda sunucu bütün çözüm ağını, *freeze* komutunu yayarak dondurur. Çözüm ağının donması sağlandıktan sonra bütün istemcilere *clean* iletisi gönderilir. *clean* iletisini alan bir istemci, o anki çözümden kalan bütün bilgileri ve mesajları siler. Bu durumu belirtmek için sunucuya ayrıca bir mesaj gönderilmez. Sunucu da

bütün istemcilere clean mesajını gönderdikten sonra kendi içindeki bilgi ve mesajlarını temizler. Artık bütün çözüm ağı temizlenmiş bir biçimde yeni çözümü beklemektedir. Görüldüğü gibi buradaki en büyük yük *freeze* işlemi üzerinde bulunmaktadır.

4.16. Çözüm Ağında Düşme

Klasik geriye-iz-sürme (bactracking) kavramı concurrent Prolog için mevcut değildir. İndirgeme (reduction) sırasında herhangi bir istemcide *fail* durumu ile karşılaşılabilir. Bu durumda *fail* üreten istemci *fail*'e sebep olan çağrıyı fail(Goal) terimi ile sunucuya gönderir. Bu düşme durumunda bütün çözüm ağının hesaplamayı durdurması gereklidir.

fail(Goal) iletisini alan sunucu, *fail*'e sebep olan *Goal* ve istemciyi kendi konsoluna ikaz mesajı olarak yazar. Daha sonra, yukarıda anlatıldığı gibi, *clean* işlemi ile bütün çözüm ağını temizler ve yeni bir çözüm için hazır bekler.

4.17. Çözüm Ağının Tamamen Kapatılması

Bütün çözüm ağının durdurulması ve istemciler ile olan bütün bağlantıların koparılması istenebilir. Bu durumda sunucuya *stop* mesajı girilir. *stop* iletisini alan sunucu bütün istemcilere *stop* iletisi gönderir. *stop* iletisini alan istemci, sunucu ile olan TCP/IP bağlantısını koparır. Sunucu da aradaki bütün bağlatıları ve kendi içindeki sunucu soketini kapatır. Sunucu ve istemciler, bütün bağlantılarını kopardıktan sonra kabuk durumuna (shell prompt) düşerler.

4.18. "Top Level" Durumuna Düşmek

Hem sunucu hem de istemcilerde klavyeden sisteme ileti girilebilir. Klavyeden girilen top_level komutu ile ilgili proses klasik Prolog durumuna düşer. Bu sayede kullanıcı, programın o anki bütün durumlarını standart Prolog komutları ile izleyebilir.

top_level komutunu alan bir proses o anki bütün işlemlerini durdurur ve kullanıcı klasik *?-* durumuna düşer. *Ctrl-D* tuşuna basılarak prosesin kaldığı yerden çözüme devam etmesi sağlanır. Bu tür bir işlem bütün çözüm ağı dondurulduktan sonra yapılmalıdır.

4.19. Sunucunun İstemcilerden İleti Toplama Yöntemi

Çözüm ağındaki istemciler kendi aralarındaki bütün ileti alışverişini sunucu aracılığı ile yaparlar. Çözüm ağının ayağa kalkması için önce sunucunun ayağa kalkması gerekir. Ayağa kalkan sunucu önce bir sunucu soketi (server socket) açar ve bu soket üzerinde bağlantı isteklerini beklemeye başlar.

Bağlantı talebinde bulunan istemci ile sunucu arasında iki yönlü soket bağlantısı (bidirectional socket connection) kurulur. Bağlantı kuran istemci ivedilikle kendi *alias* adını gönderir. Bu adımdan itibaren artık bu istemci çözüm ağının bir parçası olmuştur. Sunucu kendisine bağlanması gereken bütün istemcilerin adlarını bilmektedir. Bu adlar concurrent Prolog programı derlenirken önişlem sırasında tespit edilmektedir.

Bütün istemciler bağlantı isteklerini gönderdikten ve bağlantılarını tamamladıktan sonra, sunucu çözümü başlatmak için artık hazırdır. Ayrıca sunucuya bağlı olan klavye de ayrı bir istemci olarak değerlendirilebilir. Bir istemci gibi aynı şekilde sunucuya ileti gönderebilir. Fakat buradaki veri trafiği tek yönlüdür. Aynı zamanda klavyenin bir *alias* adı mevcut değildir. Bu sayede programın genel

yapısına hiç dokunmadan kullanıcı ile istemci arasında etkileşimli bir çalışma ortamı sağlanmış olur.

Bütün sistem için bir adet kuyruk (queue) kullanılmıştır. Aslında gelen mesajların tipine ve önem sırasına göre birden fazla öncelikli kuyruk (priority queue) kullanılmalıdır. Bu da GC ile beraber, ileride yapılması gereken en önemli iyileştirme çalışmalarından biri olacaktır.

Sıfır numaralı soket her zaman klavye girişleri için kullanılmaktadır. Diğer bağlantılar ise genelde dört numaralı soketten başlar. Sunucuda bulunan bir Prolog yüklemi *giriş motoru (input engine)* görevi görür. Bu yüklem soket bağlantılarında bulunan bütün iletileri toplar ve sunucu giriş kuyruğuna (server input que) yerleştirir. giriş motoru, *select()* sistem fonksiyonunu kullandığından soketlerde bilginin olup olmadığı çok kısa bir zaman alır. Diğer bir deyişle soket kontrolü sunucunun genel işleyişine sekte vurmaz. Aynı zamanda bu bekleme süresi kullanıcı tarafından denetlenebilir. Bu denetleme sayesinde sunucunun gereksiz yere çalışması engellemiştir. Bu süreyi *TimeOut* süresi olarak adlandıralım.

Eğer *TimeOut* süresi *0* olursa, en az bir sokete bilgi gelene kadar bütün sunucu askıya alınır, *bloke* olur. Fakat bu durum her zaman istenen özellik değildir.

Çalışma sırası giriş yüklemine geldiğinde bütün sistem askıya alınmış olsun. Bu sırada, sunucunun istemcilere göndermesi gereken iletiler bulunabilir. Bu gönderim işlemi de gecikmiş olur. Bu istenmeyen bir durumdur. Bundan dolayı *TimeOut* değeri her zaman sıfıra en yakın *float* sayı olarak seçilir. Böylece çok kısa bir zaman aralığında soket denetimi yapılır ve sunucunun yapması gereken diğer işlemler için denetim tekrar diğer yüklemlere aktarılır.

Sunucu tarafında, sokete bilgi gelmesini beklemekten başka yapacak başka bir iş olmayabilir. Bu durumda *TimeOut* değeri, programın içinde otomatik olarak *0* yapılır. Böylece sokete bilgi gelene kadar bütün sunucu uykuya dalar. Sokete bilgi gelir gelmez sistem uyanır ve tekrar *TimeOut* değeri kullanıcının vermiş

olduğu değere getirilir. Bu basit yöntem sayesinde istemcilerde ve sunucularda çok büyük bir CPU tüketimi azalışı olmuştur. Bu yöntem uygulanmadan önce %100 civarında olan CPU tüketimi, %1'in altına düşmüştür.

Bu yöntemdeki en büyük problem istemci veya sunucunun, sokette bilgi beklemenin dışında başka hiçbir işinin olup olmadığının etkin bir biçimde tespit edilmesidir. *mark* yöntemi ile bu problem çözülmüştür. Bu yöntem Shapiro tarafından ölümcül kilitlenme (dead lock) olup olmadığının kontrolü için kullanılmıştır. Bu yöntemin Berk Sistem'de uygulanması ayrı bir başlık altında anlatılacaktır.

4.20. İletilerin İşlenme Tekniği

Hem istemci hem de sunucuda istemcilerin işlenme tekniği aynıdır. *Giriş motoru*, soketlerden prosese gelen bütün iletileri toplar. *Process Engine* ise giriş kuyruğundan bir eleman seçer. Eleman işlendikten sonra elde edilen sonuçlar veya bilgiler standart Prolog'un kendi heap alanı içinde saklanabilir. Ya da elde edilen bir sonuç doğrudan ilgili istemci veya sunucuya gönderilebilir.

Diğer prosese aktarılacak bilgiler doğrudan gönderilir, burada ayrıca bir *çıkış motoru* gibi bir kavram kullanılmamıştır. Son olarak *Process Engine* kendi giriş kuyruğuna da ileti ekleyebilir. Böylece giriş kuyruğu üç farklı yerden beslenmektedir. Bunlar sırasıyla; prosesin kendi içinden, diğer bilgisayarlardan veya kullanıcının klavyesindendir.

İki numaralı değişkenin bir an gelmiş olduğunu varsayalım. need(2, fulcrum) iletisi PE (prosess engine) tarafından giriş kuyruğundan çekilir. *2* numaralı değişken gelmiştir. Bu değişken hemen term(Term, Dict) yapısı ile *fulcrum* adlı istemciye gönderilir.

Bir diğer durum ise giriş kuyruğunun klavye tarafından beslenmesidir. Klavyeden *clean* mesajının girildiğini kabul edelim. Bu durumda sunucu bütün istemcilere

clean mesajını göndermektedir. İşlem sonucunda kuyruk boyu *1* ileti kısalmıştır. Bu örnekte bir proses birden fazla prosese ileti göndermiştir. Berk Sistem'de ileti göndermek için ayrı bir motor kullanılmamıştır. İlgili mesajlar sıra ile gönderilecek yerlere gönderilmektedir.

Bir proses kendi giriş kuyruğunda bulunan iletilere göre davranış gösterir. Bunun dışındaki hiçbir etki prosese iş yaptıramaz. Bu uygulanan yöntem, proses işleyişlerine büyük bir sadelik ve basitlik getirmiştir. Sisteme yeni bir iletinin ve davranışının eklenmesi veya çıkarılması, diğer bir deyişle sistem modülerliği çok üst seviyelere çıkartılmıştır. Ayrıca bu modüler yöntemin *mark* iletisi ile beraber kullanılması sayesinde CPU kullanım oranı çok fazla azaltılmıştır.

4.20.1. "mark" iletisi

Bazen öyle bir durum gerçeklenir ki giriş kuyruğu içindeki iletiler, proses içinde hiçbir değişikliğe sebep olmadığı gibi diğer proseslere de bilgi göndermezler. İşte böyle bir durumda işlenmek üzere giriş kuyruğundan çekilen her bir eleman tekrar giriş kuyruğuna eklenir. Bu özel halde prosesin *durumunu (state)*, sadece dışarıdan gelen bir ileti değiştirebilir. İşte bu özel durumda iletilerin işlenmesinin dışarıdan bilgi gelene kadar durdurulması gerekir. Unix'deki select() fonksiyonun *TimeOut* değeri *0* yapılarak bütün sistemin bloke olması kolaylıkla sağlanır.

Bu özel halin tespiti için *mark* iletisi kullanılmıştır. Bu ileti kuyruğun en devamlı iletisidir. Hiç bir zaman kuyruktan ayrılmaz. Ayrıca bu ileti ile birlikte bir adet de *flag* değişkeni kullanılmıştır.

mark iletisi her zaman PE tarafından üretilir ve giriş kuyruğuna atılır. Bu ileti kuyruğa her zaman *flag=0* yardımcı değeri ile beraber girer. *flag*, global bir değişken olup, sistemin o anki durumunun bir önceki duruma göre değişip değişmediğini gösterir.

Örneğin bir *indirgeme* işlemi başarılı olursa sistemin o anki durumu değişecektir. Bu durumda indirgemeyi yapan yüklem *flag* değerinin 1 yapılmasından sorumludur.

mark iletisi PE tarafından işlenirken iki ayrı durum söz konusudur. Eğer *mark* iletisi işleneceği zaman *flag* değeri *0* ise giriş kuyruğundaki hiçbir ileti sistemin o anki durumunu değiştirmemiştir. Bu durumda aynı iletileri tekrar tekrar gözden geçirmenin hiçbir anlamı yoktur. Çünkü sistemin durumunu değiştirmeyecekleri kesindir. İşte bu sebeten dolayı PE derhal bütün sistemi bloke eder ve sokete bilgi gelmesini bekler. Sokete bilgi geldiği an ileti işleme çevrimi devam edecektir.

Diğer bir ihtimal ise *flag* değişkeninin *1* değerini almasıdır. Bu durumda *mark* mesajına kadar olan iletileriden birisi *mark* değişkenine *1* değerini atar. Diğer iletilerin bu değişimden etkilenme ihtimali vardır. Bundan dolayı sistem bloke edilmez ve *mark* iletisi *flag=0* değeri ile tekrar giriş kuyruğuna eklenir.

Giriş kuyruğunda bulunan hiçbir ileti sistemin o anki durumunu değiştirmemiş olabilirler. Fakat herhangi bir anda sokete gelen bir bilgi *flag* değerini hemen *1* yapar. Böylece sistemin durumu *değişmiş* olur.

4.20.2. "nop" iletisi

Tamamen bloke olmuş olan bir sistemi kısa bir süre de olsa aktif hale getirmek isteyebiliriz. Bu durum için *nop (no operation)* iletisi tanımlanmıştır. Bu iletinin hiçbir etkisi mevcut değildir, daha çok debug amacı ile kullanılmaktadır. *nop* iletisi genelde klavyeden girilir. Klavyeden girilen her ileti bloke olmuş sistemi hemen uyandırır. Bloke olmuş sistem kısa bir süre çalışır ve soketlere herhangi bir yeni bilgi ulaşmamış ise tekrar bloke olur.

4.20.3. "shell/1" iletisi

Kullanıcı, sunucu veya istemciler çalışırken işletim sistemine *shell/1* iletisi ile komut gönderebilir. Komut işletim sisteminde yürütülür. Komut *arka planda (backgound)* çalışacak şekilde verilmeyebilir. Bu durumda komutun yürütülmesi bitene kadar istemci veya sunucu işlemleri bloke olacaktır.

4.20.4. "debug" iletisi

Bu ileti ile hem sunucu hem de istemciler tarafında bütün çözüm adımları takip edilebilir. Debug işlemi ile aşağıdaki bilgiler gözlemlenebilir.

İstemciden sunucuya ve sunucudan istemciye gönderilen bütün iletiler izlenebilir.

Giriş kuyruğundaki (input que) bütün iletiler izlenebilmektedir.

İstemcilerde bulunan *UnboundedDict* tablosu izlenebilmektedir. Bu seçenek sunucu için geçerli değildir, çünkü sunucu içinde böyle bir tablo bulunmaz.

İstemci içinde yapılan her türlü indigeme işlemi ayrıntılı bir biçimde izlenebilir.

İstemci içinde herhangi bir tabloya yapılan bir güncelleme izelenebilir.

Sistem bütün çözümü bitirdiğinde elde elde en son tabloları listeleyebilir.

Sunucu veya istemci bloke olduğunda uyarı mesajının verilmesi sağlanır.

Bütün çözüm ağı herhangi bir amaç için *freeze* durumuna getirildiğinde uyarı mesajı verilmesi sağlanır.

Yukarıdaki seçeneklerden herhangi birini aktif hale getirebilmek için derleme veya yürütme zamanında gerekli atamalar yapılabilir.

4.21. İndirgeme İşlemi

Bir çağrı yapıldıktan sonra yürütülen işlemlere indirgeme işlemi denir. İndirgeme işlemi, baş birleştirmesi, *guard* hesaplaması ve gövde çağrılarının paralel olarak yürütülmesi için yapılması gereken işlemlerden oluşur. Bu bölümde Berk Sistem'in indirgeme tekniği anlatılacaktır.

4.21.1. Ön Hazırlık

İndirgeme işlemi sadece istemci tarafında gerçeklenir. İndirgenecek çağrı için mevcut olan şartların tamamının gerçeklenip gerçeklenmediği bu adımda kontrol edilir. Şartlar gerçekleşene kadar çağrı indirgemesi askıya alınır.

İndirgenecek bir çağrı giriş kuyruğuna her zaman call4(NeedAnd, NeedOr, WaitAnd, WaitOr, Goal, Share) iletisi ile girer. Bu ileti aslında kullanıcının yazmış olduğu concurrent Prolog kodunun önişlemden geçmiş halidir. Bu cümleler istemci tarafında *berk_fact/3* yapıları içinde saklanırlar.

call4/6 yapısında bulunan ilk *4* argüman, bir çağrının indirgenmesi için gerekli olan şartları kesin olarak temsil ederler. Çağrının indirgenmesi için *NeedAnd, NeedOr, WaitAnd, ve WaitOr* şartlarının sağlanması gerekir. Diğer bir deyişle bütün şartlar *AND* mantıksal operatörü ile birleştirilmiştir. Programcı bu şartları hangi sırada yazarsa yazsın her zaman gösterilen sırada kontrol edilir. Ayrıca değişkenlerin tutarlı bir biçimde listelerin içine yerleştirilmesi programcının sorumluluğuna terkedilmiştir. Berk Sistem içinde, şartlar için tutarlılık denetimi yapılmaz. İleriki çalışmalarda bu analiz eklenebilir.

İndirgeme şartlarını temsil eden ilk dört argüman bir tamsayı listesinden oluşur. Programcının NeedAnd=[1, 4], NeedOr= [1] gibi bir şart yazması aslında gereksizdir. *NeedAnd* şartı sağlandığı için *NeedOr* şartı da sağlanmış olur.

Genellersek, *NeedAnd* içinde bulunan herhangi bir değişken *NeedOr* listesi içinde bulunamaz. Örnek olarak

NeedAnd=[1,4], NeedOr= [], WaitAnd=[], WaitOr= [2,5,7]

listelerini gözönüne alalım. Çağrının indirgenmesi için önce *1* ve *4* numaralı değişkenler sunucudan gelmiş olmalıdır. Eğer sunucudan talep edilmeyen değişken varsa talep edilir. Bu indirgeme işlemine gelmeden evvel *1* numaralı değişken sunucudan talep edilmiş olsun. Bu durunda sadece *4* numaralı değişken sunucudan talep edilecektir.

1 ve *4* numaralı değişken istemciye ulaştıktan sonra *NeedOr* şartına bakılır. Burada herhangi bir şart mevcut değildir. Benzer şekilde *WaitAnd* listesi de boş olduğu için sonraki *WaitOr* listesine geçilir. *2*, *5* veya *7* numaralı değişkenlerden herhangi biri sunucuya ulaşmış ise bu şart da gerçekleşmiş olur. Artık bu adımdan sonra indirgeme işlemine geçilir.

NeedAnd veya *NeedOr* listeleri içindeki her bir değişken mutlaka sunucudan talep edilir. *WaitAnd* ve *WaitOr* listeleri içindeki değişkenler sunucudan talep edilmez. *Wait* listeleri içindeki değişkenler için ise körleme bir biçimde bekleme yapılır.

Körleme beklenen değişkenlerin, bekleyen istemciye ulaştırılması, değişkeni üreten prosesin sorumluluğu altındadır. Körleme değişken üreten prosesler *share* listesi yardımı ile istemcilere değişken gönderir. Bu gönderim işlemi tamamen programcının sorumluluğu altındadır.

call4(NeedAnd, NeedOr, WaitAnd, WaitOr, Goal, Share)

yapısı 6 argümanlı bir giriş kuyruğu (input queue) iletisidir. Bu ileti giriş motoru tarafından kuyruktan çekilir (dequeue). *NeedAnd* listesi içindeki herhangi bir değişken gelmiş ise, bu değişken hemen listeden atılır.

Örneğin NeedAnd= [1, 4] olsun. Sadece *4* numaralı değişken gelmiş ise *NeedAnd* listesi *[4]* olarak güncellenir ve *call4/6* iletisi tekrar giriş kuyruğuna atılır. Bir sonraki çevrimde işlem sırası yine bu iletiye geldiğinde, bir an için *1* numaralı

değişkenin de sunucuya gelmiş olduğunu varsayalım. Bu durumda *NeedAnd* listesi *[]* (boş liste) değerini alacaktır. Bir liste eğer *[]* değerini alırsa derhal bir *alt call* yapısına düşülür. Diğer bir deyişle *call4/6* iletisi *call3/5* iletisi haline getirilir. Bu yeni ileti *call3(NeedOr, WaitAnd, WaitOr, Goal, Share)* yapısı ile takrar giriş kuyruğuna yeni bir ileti olarak atılır.

NeedOr listesi içindeki en az bir değişken gelmiş ise, *call3/5* iletisi *call2(WaitAnd, WaitOr, Goal, Share)* yapısı ile tekrar giriş kuyruğuna atılır. Benzer biçimde *WaitAnd* ve *WaitOr* listeleri de işlendikten sonra giriş kuyruğuna *call0(Gaol, Share)* nihai yapısı atılır.

Özetle, giriş kuyruğunda bulunan *call4(NeedAnd, NeedOr, WaitAnd, WaitOr, Goal, Share)* iletisi, önişleme sırasında, sıra ile

```
call3(NeedOr, WaitAnd, WaitOr, Goal, Share)
call2(WaitAnd, WaitOr, Goal, Share)
call1(WaitOr, Goal, Share)
call0(Goal, Share)
```

iletileri haline gelirler. *call4/6* yapısından *call0/2* yapısına ulaşma işlemine *indirgeme önhazırlığı* diyeceğiz. Önhazırlıktan geçen bütün çağrılar artık indirgenmeye adaydır.

4.21.2. İndirgeme

İndirgenecek çağrı, giriş kuyruğunda her zaman *call0(Goal, Share)* şeklinde bulunur. Bu iletiye indirgeme önhazırlığından geçildikten sonra ulaşılabilir.

İndirgeme (reduction) işlemi, bir çağrı için uygun cümlenin seçimi ile yapılır.

Bu temel işlemi sadece istemciler icra ederler. İndirgeme, bir istemcinin CPU tüketiminin çok büyük bir kısmını oluşturur. İndirgeme işlemi aşağıdaki adımlar gerçeklenerek yapılır.

İndirgenecek çağrıyı f/n ile temsil edelim. f ve n harfleri standart Prolog'daki gibi, functor ve argümanları (artiy) temsil eder.

Önce paralel kod ile yazılmış bütün cümleler içindeki f/n'ler seçilir.

Sonra f/n cümlelerinin baş kısımları f/n ile klasik anlamda unify (MGU, Most General Unifier) edilir. İlk yazılan cümle en önce, son yazılan cümle en son incelenir. Diğer bir deyişle kullanıcının yazdığı sırada (text order) inceleme yapılır. Kullanıcının yazmış olduğu cümleleri aşağıdaki gibi f_i/n ile temsil edelim.

$$
\begin{array}{ll}
f_1/\mathrm{n} & \leftarrow Guard_1 \mid Body_1. \\
f_2/\mathrm{n} & \leftarrow Guard_2 \mid Body_2. \\
 & \vdots \\
f_n/\mathrm{n} & \leftarrow Guard_n \mid Body_n.
\end{array}
$$

f/n ile f_i/n yapıları sıra ile unify edilir. Eğer hiçbir cümle seçilemez ise hesaplama başarısız olur (computation fails) ve sunucuya "*fail*" iletisi gönderilir.

Diğer durumda ise f/n yapısı herhangi bir f_i/n ile başarılı bir biçimde birleştirilmiş (unify succeeds) olsun. Baş birleştirme işleminden hemen sonra bu cümlenin $Guard_i$ ifadesi yürütülür. Bu kısım tamamen standart Prolog terimlerinden oluştuğundan, $call(Guard_i)$ ile indirgeme yapılır. Eğer bu call işlemi başarısız olursa bir sonraki cümleye geçilip tekrar *head unify* yapılır. Bir an için $Guard_i$'nin de başarılı bir biçimde hesaplandığını varsayalım. İşte yüklemin bu cümlesi artık seçilmiştir (comitted clause). *Guard*'dan hemen sonra yazılan *commit* operatörü aslında klasik bir *cut* operatörü gibi çalışır ve bütün alternatif çözümleri (choice point) yok eder. $Body_i$ elemanları $call4/7$ yapılarından oluşmuş ve önişlem sırasında kurulmuştur.

Gövdenin işlenmesi pek çok özellik içerdiğinden ayrı bir başlık altında incelenecektir. Aşağıdaki paragraflarda konu kısaca anlatılmıştır.

Yerel makine içinde değil de diğer makine içinde indirgenecek çağrıları uzak çağrı, bunun dışındaki çağrıları da yerel çağrı olarak adlandırmıştık. Uzak çağrılar sunucu aracılığı ile aday istemciye gönderilecektir.

Gövde içinde bulunan her bir *call4/7* yapısı iki ayrı gruba ayrılır. Birinci grupta uzak çağrılar mevcuttur. Diğer bir deyişle bu prosesler diğer istemcilerde indirgenecektir. *call4/7* yapısı, *call4/6* yapısı ile benzerlikler gösterir. Son argüman olarak indirgemenin yapılacağı istemcinin adı mevcuttur. *call4/7* yapıları ilgili istemcide indirgenmesi amacı ile sunucuya gönderilir. Sunucu da aldığı bu iletinin son argümanından kime gönderileceğini anlar ve bu son argümanı silerek, ilgili yapısı *call4/6* yapısına çevirerek istemciye gönderir. İstemcinin *call4/6* yapılarını indirgemesi daha önceki paragrafta anlatılmıştı.

İkinci grupta ise yerel makinede indirgenecek çağrılar mevcuttur. Bu ikinci grubun bütün iletileri *call4/6* olarak giriş kuyruğuna atılacaktır. Görüleceği gibi bir prosesin kendi giriş kuyruğuna *call4/6* iletisini göndermesi ile sunucudan indirgemek amacı ile gelen *call4/6* prosesinin işlenmesi arasında bir fark yoktur.

İndirgencek çağrı her zaman *call0(Goal, Share)* yapısında ulaşmak zorundadır. *call0/2* iletisi ana indirgeme işlemini başlatır. Asıl amaç her zaman *call0/2* yapısına ulaşmaktır. Buraya ulaşıldığında artık *Goal* içinde beklenilen hiçbir değişken mevcut değildir, hepsi istemciye ulaşmıştır.

Share listesi, *Goal* indirgendikten sonra sunucuya gönderilecek ortak değişkenlerin numaralarını barındırır. *Share* listesi, [1, 2, 3=fulcrum, 4, 5] şeklinde *No* veya *No= Alias* elemanlarından oluşur. Eğer *No* hesaplanmış ise hemen sunucuya gönderilir. Hesaplanmamış ise gövde içinde bulunan *call/_* komutlarının *share* listelerine atanırlar. Çünkü bu değişkenin orada hesaplanma olasılığı her zaman mevcuttur. Bu örnekte, eğer *1* veya *2* numaralı değişken hesaplanmış ise hemen sunucuya gönderilir. *3* numaralı değişken hesaplanmış ise bu değişken dondurulur ve sözlüğü ile beraber sunucuya gönderilir. Fakat bu değişken sunucuya gönderilirken var(3-fulcrum, term(p(X, Y), ['5'=X, '6'=Y])) örneğinde olduğu gibi *Alias* bilgisi ile beraber gönderilir. Daha önceki yapıdan farklı olarak

fulcrum adındaki *Alias* bilgisi eklenmiştir. Sunucu kendine gelen bu terimi saklar, aynı zamanda *Alias* adlı istemciye yollar.

Local makinenin *phantom* olduğunu varsayarak, call0(p(X, Y), [1, 2= tornado]) iletisini ele alalım. *p/2* cümlesi berk_fact(Head, Guard, Body) içinden alınacaktır. Örnek bir *berk_fact* olgusu aşağıda verilmiştir.

```
berk_fact  ( p(a, X),
             q(X),
             [ call([X], [], [], [], r(X, Y), [Y], fulcrum),
             call( [], [],[Y], [], s(Y), [], phantom)
             ]
             ).
```

Dikkat edilirse bütün liste elemanları *değişkendir.* Dondurma işlemi sırasında bu değişkenlere sayı atanacaktır. Bu işlem gövdenin işlenmesi sırasında yapılacaktır.

4.21.3. Gövdenin İşlenmesi

İndirgeme aşamalarının en karmaşık ve çok vakit alan kısmı gövdenin işlenmesidir. Bu işlemlerin ana indirgeme adımları aşağıda verilmiştir.

Goal adındaki çağrının indirgeneceğini, ve *Share* listesinin de *Goal* listesi ile kullanılacağını varsayalım. Diğer bir deyişle *call0(Goal, Share)* iletisinin indirgeneceğini varsayalım.

A- Geriye-iz-sürmeyi engellemeden *berk_fact(+Goal, -Guard, -Body)* olguları üzerinde sorgulama yapılır. Eğer *fail* olursa bütün hesaplama başarısız olmuştur (computation fails). Bu durumda sunucuya *fail(Goal)* iletisi gönderilir ve istemci kendisini *freeze* durumuna sokar. Çünkü *fail* durumundan sonra istemcinin artık herhangi bir hesaplama yapması anlamsızdır. Fakat yine de sunucudan gelecek olan iletileri eksiksiz olarak kabul eder.

En az bir adet *berk_fact/3* olgusunun *berk_fact(Goal, Guard, Body)* olgusu ile başarıyla birleştirildiğini (unify) varsayalım.

B- *call(Guard)* çağrısı yapılır. *Guard* ifadesi standart Prolog çağrılarından oluşur. *call/1*'de standart Prolog fonksiyonudur. Eğer bu çağrı *fail* olursa *A*'dan geriye-iz-sürme yapılır.

C- Bu adım indirgemenin başarıya ulaştığı noktadır. Artık bu cümle seçilmiştir. Diğer bütün *berk_fact/3* olgularını ve bütün CP'leri yok etmek için kesme işareti, '!' uygulanır. *commit* operatörü burada aynen bir '!' işareti gibi çalışır.

D- *berk_fact/3*'den gövde ifadesi elde edilir. *berk_fact/3* yapısı önişleme sırasında elde edilmiş idi. Gövde, bir listedir ve *call/7* elemanlarından oluşur. Örneğin, Body= [call([X], [], [], [], r(X, Y), [Y], fulcrum), call([], [],[Y], [], s(Y), [], phantom)] şeklinde bir liste elde edilir. *call/7*'nin ilk *4* elemanı *need* listeleridir. Sonraki argüman gövdenin bir çağrısını oluşturur. Altıncı argüman *share* listesini ve son argüman ise, 5. argümandaki çağrının indigeneceği istemcinin adını gösterir. Sembolik olarak *call/7* yapısı call(NeedAnd, NeedOr, WaitAnd, WaitOr, BodyGoal, ShareList, ClientName) ile gösterilebilir.

E- *ClientName*, yerel makinenin adı ile aynı ise bu gövde çağrısına, yerel çağrı, diğer gövde çağrılarına da uzak çağrı demiştik. Önce gövde içinde bulunan yerel ve uzak çağrılar ayrı ayrı sınıflandırılırlar. Örneğimizdeki yerel ve uzak çağrılar aşağıdaki gibi gösterilebilir.

Local= [call([X], [], [], [], r(X, Y), [Y])]

Remote= [call([], [],[Y], [], s(Y), [], phantom)]

Yerel (local) makine adının *fulcrum* olduğu varsayılmıştır. Dikkat edilirse *Local* listenin elemanları *call/6* yapılarından oluşur. Çünkü bu listenin elemanları, yerel makinede indirgenecektir. *Remote* liste içindeki her bir elemanın hangi makinede indirgeneceği son argümanda tespit edilmiştir.

Örnekte, s(Y) çağrısı *phantom* adlı makinede indirgenecektir. r(X, Y) çağrısı ise yerel makinede indirgenecektir.

F- *Unbounded dict* ve *global dict* kullanarak *Local* ve *Remote* listelerinin içindeki elemanlar tek tek dondurulur. Ayrıca her bir *call/n* yapısını *call4/n* yapısı şekline çevir. Bu *functor* değiştirme işlemi, ilgili çağrının doğrudan istemci kuyruğuna girebilmesi için bir ön hazırlıktır. Örnekteki, *Local* ve *Remote* listeleri aşağıdaki şekle bürünecektir.

```
Local = [call4(['1'], [], [], [], r(X, Y), ['2'])]
Remote= [call4( [], [],['2'], [], s(Y), [], u2)]
```

G- *Share* listesindeki değişkenleri, *bounded* ve *unbounded* olarak ikiye ayrılır. *unbounded* değişkenlerden oluşan listeyi *unbounded*, diğer listeyi ise *bounded list* olarak adlandıralım.

H- *Bounded* değişkenlerin bulunduğu listeyi dondur ve hemen kendi sözlüğü ile beraber sunucuya gönder.

I- *Unbounded* listesi dolaşılır. Buradaki her bir değişken mutlaka *Local* ve *Remote* listeleri içine aşağıdaki gibi gömülmelidir.

Local veya *Remote* içindeki her bir *call4/_* dolaşılır ve gövde terimleri ele alınır. Eğer *Unbounded* içindeki bir değişken gövde atomu içinde varsa ve gövde *goal– >share* içinde mevcut değilse bu değişken hemen *goal– >share* içine eklenir. Yeni *Local* ve *Remote* listeleri içindeki *share* listeleri artık genişlemiştir.

J- *Remote* liste hemen sunucuya gönderilir. İçine kendi sözlüğü de eklenir.

K- Yerel olanlar giriş kuyruğuna atılır.

4.21.4. Yardımcı Bir "berk_fact" Olgusu: dummy

Çeşitli amaçlarla kullanmak amacı ile dummy adlı özel bir olgu tanıtılmıştır. Bu olgu, standart Prolog'daki *true* yapısına benzer. Fakat daha esnektir. *dummy* fonksiyonu doğrudan veya dolaylı olarak kullanılabilir.

dummy@fulcrum. yazım şekli *doğrudan kullanıma* bir örnek teşkil eder. *fulcrum* adlı makinede *dummy* adlı fonksiyon işleyecektir. Bu kullanım tarzı ile, aslında *fulcrum* adlı istemciyi çözüm ağı içine katmış oluyoruz.

merge(Xs, [], Xs). olgusunu düşünelim. Bu olguda *Xs* değişkeninin ortak kullanılabilmesi için *dummy* fonksiyonu doğrudan

```
merge(Xs, [], Xs) :- dummy with share(Xs).
```

şeklinde olgu içine katılabilir.

Ayrıca *true* olgusunun uygulanmasında ve unify işlemlerinde de *dummy* fonksiyonu dolaylı yollardan kullanılacaktır.

dummy fonksiyonu dummy ← true | true. şeklinde tanımlanmıştır. Bu cümle bütün istemcilerde berk_fact(dummy, true, []). şeklinde bulunur.

4.21.5. "fail" fonksiyonu

Klasik *fail* fonksiyonu Berk Sistem'de de tanımlanmıştır. Fakat daha geniş bir yan etkisi mevcuttur. Klasik Prolog'da kullanılan *fail* fonksiyonu, Prolog sistemini *backtrack* yapmaya ve yeni bir çözüm bulmaya zorlar. Fakat concurrent Prolog'da backtrack kavramı mevcut değildir. Bundan dolayı *fail* fonksiyonu bütün hesaplamayı çökertmek için kullanılacaktır. Bunun için sunucuya fail(fail) iletisi gönderilir ve istemci kendini *freeze* durumuna geçirir. En genel durumda fail(Goal) ile sunucuya *fail* durumuna düşüldüğü bildirilir. Burada Goal= fail olarak ele alınmıştır. Örneğin;

```
p ← q | fail.
```

4.21.6. "true" Fonksiyonu

Standard Prolog'daki *true* fonksiyonu ile tamamen aynıdır. call0(dummy, Share) iletisi kurularak uygulanır. Örnek;

```
dummy(Xs, [], Xs) :- true with share(Xs).
```

4.21.7. Klasik Prolog ile Etkileşim

Gövde içinden Prolog çağrıları yapılabilir. Bu çağrıları diğer Berk Prolog çağrılarından ayır edebilmek için *Prolog/1* yapısı kullanılmıştır.

Örnek:

```
p ← q |
        r(X)@fulcrum with share(X),
        prolog(assertz(X))@phantom with need(X).
```

Yukarıdaki *p/0* örneğini ele alalım. Burada, önce q çağrısı icra edilecektir. Eğer q başarılı olursa r(X) çağrısı *fulcrum* adlı istemcide işlenecektir. prolog(assertz(X)) çağrısı da *phantom* adlı makinede işlenecektir. *phantom* işlemcisine X değişkeninin değeri ulaştığında *prolog/1* çağrısı yapılacak ve *assertz/1* standart Prolog fonksiyonu icra edilecektir.

Prolog/1 çağrısı aşağıdaki gibi uygulanır. Bu çağrı her zaman call0(Prolog(Goal), Share) iletisi içinde bulunur.

- call(Goal) yap. Buradaki *call/1*, standart Prolog fonksiyonudur.
- *call/1* başarısız olursa sunucuya fail(Goal) gönder ve *freeze* durumuna geç.
- *call/1* başarılı olursa call0(dummy, Share) iletisini kur ve işle.

4.21.8. "unify" Fonksiyonu

Klasik Prolog'daki *unify* fonksiyonu Berk Prolog'da da geçerlidir. Bu fonksiyon doğrudan Left= Rigt şeklinde gövde içinde kodlanır. *unify* fonksiyonu

```
unify(X, X) ← true | true.
```

cümlesi ile temsil edilir. Bu cümlenin *berk_fact* olgusu berk_fact(unify(X, X), true, []). şeklinde tanıtılmıştır. Bu olgu bütün istemcilerde istisnasız bulunur ve aşağıdaki gibi uygulanır.

- call0(Left= Right, Share) iletisi call0(unify(Left, Right), Share) iletisine çevrilir ve işlenir.

Örnek:

```
p ← q | ( X=r(Y) )@fulcrum with need([X, Y]).
```

4.21.9. "enque" Kavramı

Herhangi bir *f/n* çağrısını göz önüne alalım. *true/0* veya *dummy/0* değilse, bu çağrının her zaman *fail* durumuna düşme olasılığı mevcuttur. Fakat *enque* kavramı ile *f/n* çağrısının belirli bir adımdan sonra hiç *fail* olmamasını sağlayabiliriz. Bunu sağlamak için *guard* içinde *enque* fonksiyonu kullanılır. Eğer yürütme, *enque* fonksiyonuna kadar ulaşmayı başarmış ise, *f/n* çağrısı kesinlikle *fail* olmaz. Bu fonksiyondan sonraki bütün cümlelerin baş birleştirme işlemleri ve *guard çağrıları* başarılı olmasa bile, bu *f/n* çağrısı hiç *fail* olmaz. Sanki hiçbir şey olmamış gibi bu çağrı tekrar yürütme kuyruğuna atılır, sırası geldiğinde takrar icra edilir. Tekrar kuyruğa giriş olduğu için bu fonksiyona *enque* denilmiştir. Bu fonksiyon aslında *f/n* fonksiyonuna bir tür aşı yapar, ve artık *fail* durumuna karşı korunmuş olur.

Bir *guard* içinde birden fazla *enque* veya bir yüklemde birden fazla *enque* mevcut ise sadece ilk *enque* dikkate alınır. Diğerlerinin bir etkisi mevcut değildir.

n adet f_i/n cümlesi temsili olarak aşağıdaki gibi gösterilsin.

$$
\begin{array}{ll}
f_1/\mathrm{n} & \leftarrow Guard_1 \mid Body_1. \\
f_2/\mathrm{n} & \leftarrow Guard_2 \mid Body_2. \\
f_3/\mathrm{n} & \leftarrow call_1, call_2, enque, call_3, call_4 \mid Body_3 \\
 & \vdots \\
f_{m-1}/\mathrm{n} & \leftarrow Guard_{m-1} \mid Body_{m-1}. \\
f_m/\mathrm{n} & \leftarrow Guard_m \mid Body_m.
\end{array}
$$

f_3/n cümlesinin *guard* çağrıları içinde *enque* bulunsun. f_3/n cümlesinden evvel hiçbir *enque* fonksiyonu bulunmasın.

f/n çağrısı yapılmış olsun. f_1/n ve f_2/n cümlerinin bir an için seçilmediğini varsayalım. f/n ile f_3/n başları başarılı bir biçimde birleştirilmiş olsun. Bu durumda yürütme, f_3/n cümlesinin *guard*'ına geçer. Eğer $call_1$ ve $call_2$ başarılı olursa yürütme *enque* fonksiyonuna geçer. İşte bu fonksiyon f/n çağrısını aşılar. Artık $call_3$ veya $call_4$ başarısız olsa bile, veya bundan sonra hiçbir f_i/n cümlesi seçilmese bile, sanki üzerinde hiçbir işlem yapılmamış gibi, f_n çağrısı tekrar yürütme kuyruğuna atılır.

enque fonksiyonu *catch all* işleminin gerçeklenmesi için de kullanılabilir. Eğer bir yüklemin en son cümlesi

$$f_n/\mathrm{m} \leftarrow \text{enque, fail} \mid Body_m$$

şeklinde yazılırsa, bu yüklem hiçbir zaman *head unify* veya *guard* çağrılarından ötürü fail olmaz. Devamlı olarak yürütme kuyruğunda kalır. Bu son cümleye gelmeden evvel bir cümle seçilir veya gövdedeki bir çağrıdan dolayı *fail* olursa bu cümle de *fail* olur.

Aslında bu gösterilen son yazım şekli

f_n/m $\leftarrow$ f_n/m

yazımının farklı bir gösterimidir.

enque fonksiyonunun uygulaması son derece basittir. Bir çağrı yapıldığında, yüklemin içinde *enque* olup olmadığına bakılmadan *global flag* değeri *0* yapılır. Herhangi bir *enque* fonksiyonuna denk gelindiğinde *enque* ile ilgili *global flag* değeri *1* yapılır. Eğer bu çağrı *fail* olursa, normal durumda fail(Çağrı) şeklindeki bir mesaj hemen sunucuya gönderilir ve bütün sistem *fail* olur. Fakat eğer *enque flag* değeri *1* ise sunucuya fail(Çağrı) mesajı gönderilmez. Bu çağrı, yürütme kuyruğuna hiçbir işlem yapılmış gibi tekrar atılır.

4.21.10. Kullanıcı Girişleri

Paralel programlar, kullanıcıdan veri girişi talep edebilirler. Bu tür bir talepte yürütme, bloke olmadan devam etmeli, kullanıcının verisi de okunabilmelidir. Bunu sağlamak için basit bir veri giriş sistemi olan *input/2* fonksiyonu tasarlanmıştır. Bu fonksiyon sistemin icrasını bloke etmeden, birden fazla veriyi aynı girişten alabilmektedir.

input(Variable, Key) fonksiyonunda *Variable* adlı değişken kullanıcıdan talep edilecektir. *Key* ise bu değişkeni temsil eden bir atomdur. Aşağıdaki basit örnekte kullanıcıdan *2* adet değişken talep edilmekte ve bu değişkenler *p/1* yüklemine, tüketilmek üzere gönderilmektedir.

```
main ←
        ...
        input(X, ad),
        input(Y, soyad),
        ...
        p(X, Y) with need_and([X, Y]).
```

input/2 fonksiyonlarında, önce veya sonra, X ve Y'den bağımsız çağrılar bulunabilir veya yürütme kuyruğu içinde icra sırası bekleyen diğer çağrılar bulunabilir. *input/2* fonksiyonlarının bütün bu çağrıları bloke etmemesi gerekir.

Örnekte *ad* ve *soyad* bilgileri kullanıcıdan talep edilmektedir. *input/2* fonksiyonları yürütüldükten sonraki herhangi bir anda, kullanıcı *ad=GorundTerm.* veya *soyad=GorundTerm.* girişini yapar. Bu girişlerde yazılan *GroundTerm* ifadeleri ilgili değişkenlere aktarılır. Buradaki *ad*, *soyad* şeklindeki atomik değerler kullanılacak olan değişken ile kullanıcının girmiş olduğu değişken arasında bağlantı kurar.

input/2 fonksiyonunun uygulanma şekli aşağıda özetlenmiştir.

- *input(Variable, Key)* yürütülecek olsun.
- *Variable* değişkenin özel numarasını elde et. Bu numara *VariableNo* olsun.
- *Key= wait(VariableNo)* şeklinde bir global flag üret.
- *dummy with wait(VariableNo)* çağrısını kur ve yürütme kuyruğuna at. Artık bu adımdan sonra *input/2* fonksiyonu *dummy/0* fonksiyonuna dönüştürülmüştür. *wait(VariableNo)* koşulu sayesinde yürütme bloke olmaz. Kullanıcı artık bu adımdan sonra *Key* adlı değişkeni girebilir.
- Kullanıcı klavyeden *Key=GorundTerm* verisini girmiş olsun. Hemen *Key= wait(VariableNo)* global değerinin olup olmadığına bakılır. Eğer bu global değer mevcut değilse bu değişkenin beklenmediği ikazı yapılır. Eğer global değer mevcut ise önce bu global değer hemen yok edilir. Böylece aynı değişkenin arka arkaya birden fazla girişi engellenmiş olur. Daha sonra kullanıcının girmiş olduğu *GroundTerm*, global sözlüğe yazılır. Fakat bu yazılma işleminden önce bu değişkenin başka bir çağrı tarafından atanıp atanmadığı kontrol edilir. Eğer atanmış ise, atanan değer ile *GroundTerm*, birleştirilir (unify). Birleştirme başarısız olursa bütün sistem *fail* olur.

- Yürütme sırası *dummy with wait(VariableNo)* çağrısına gelmiş olsun. *VariableNo* daha önceden global dict'te atandığı için *dummy/0* uyanır ve icra edilir. *dummy/0* kullanılmasının tek sebebi *input/2* içinde *share* listesinin olma olasılığıdır. Bu özel durum aşağıda verilmiştir.

```
main ←
        ...
        input(X, ad)@fulcrum with share(X),
        input(Y, soyad)@phantom with share(Y),
        ...
        p(X, Y)@tornado with need_and([X, Y]).
```

4.21.11. "if/2" Uygulaması

Klasik Prolog uygulamalarında bulunan *if/2* veya → /2 fonksiyonu aşağıdaki gibi tanıtılabilir.

```
if(A, B) ← call(A), !, call(B).
if(_, _) ← true | true.
```

Yukarıdaki tanım, bu fonksiyon için önerilen tanımlardan biridir. Başka tanımlar da mevcuttur. Fakat biz yukarıda verilen tanımı gözönüne alacağız. Buna göre *if/2* fonksiyonu Berk Prolog'da aşağıdaki gibi tanımlanmıştır.

```
% if(+, +).
if(A, B) ← call(A) | call(B).
if(_, _).
```

A ile temsil edilen argüman standart Prolog çağrısı olmak zorundadır. *A* argümanı standart Prolog'da bulunan *call/1* fonksiyonu ile çağrılır. *A* içinde Berk Prolog yüklemi bulunamaz. *B* ise Berk Prolog çağrılarından oluşur.

if/2 fonksiyonun uygulanma yöntemi aşağıda özetlenmiştir.

- *if(Cond, Body)* yürütülecek olsun.

- Klasik Prolog'daki *call/1* fonksiyonu ile *call(Cond)* çağrısı yapılır.
- *call(Cond)* çağrısı başarılı olursa, *B* gövdesine daha önce anlatılan indirgeme işlemi uygulanır. Kısaca *B* içindeki yerel çağrılar yerel makinedeki yürütme kuyruğuna eklenir. Uzak (external) çağrılar ise sunucuya gönderilir.
- *call(Cond)* çağrısı başarısız olursa, yerel yürütme kuyruğuna *true* eklenir. Bu fonksiyonun da yürütmeye herhangi bir etkisi yoktur.

Örnek:

```
main ← Guard |
       ...,
       if( (q(X), X>20),
          ( r(X)@fulcrum with share(X),
          s(X)@phantom with need(X)
          )
       ),
...
```

Yukarıda verilen kod parçasında *q/1* ve *>/2* fonksiyonları klasik Prolog fonksiyonlarıdır. *if/2* fonksiyonunun ikinci argümanındaki bütün çağrılar ise Berk Prolog çağrılarıdır. Eğer q(X), X>20 çağrıları başarılı olursa *r/1* çağrısı *fulcrum* adlı makinede ve *s/1* fonksiyonu da *phantom* adlı makinede indirgenecektir.

4.21.12. Akışların Yazılması (Stream Output)

outstream/1 yüklemi herhangi bir akışı (stream) ekrana yazmak için kullanılır. Shapiro'nun [20]'de yazmış olduğu *stream output* fonksiyonu ile birebir aynıdır. İki farklı yazım şekline sahiptir. Birincisinde akış değişkenleri körleme beklenir. İkincisinde ise değişken sunucudan aktif olarak istenir.

Körleme bekleme yöntemini kullanan *outstream* uygulaması aşağıda verilmiştir.

```
outstream([X | Xs]) ←
                    write(X)
                    |
                    outstream(Xs) with wait(Xs).

outstream([ ]).
```

Bu yönetim özellikleri aşağıda listelenmiştir.

A- İlk çağrılışta akış değişkeni *bounded* olmak zorundadır. Burada *bounded* kontrolü yapılmaz.

B- Akış içinde bulunan bütün *unbounded* elemanlar tek tek ekrana yazılır.

C- İlk *unbounded* elemana denk gelince fonksiyon kendini askıya alır.

D- *Unbounded* eleman sunucudan gelince *outstream/1* fonksiyonu tekrar uyanır ve *B* adımından itibaren yürütme devam eder.

E- *[ ]* elemanı stream'in kapatılmasına sebep olur. Bu boş küme elemanı gelene kadar stream sürekli olarak *bounded* değişken bekler.

Dikkat edilirse burada bekleme eylemi körleme yapılmaktadır. Diğer bir deyişle hangi değişkenin beklendiği sunucuya iletilmez. Değişkenin gönderilmesi, değişkeni üreten veya taşıyan yükleme aittir. Bu yöntem aşağıda anlatılacak ikinci yönteme göre daha hızlıdır. Çünkü sunucuya değişken talebi gönderilmez. Diğer yöntemin yüklemi aşağıda verilmiştir.

```
outstr([X | Xs]) ←
                 write(X)
                 |
                 outstr(Xs) with need(Xs).

outstream([]).
```

İsimlendirmenin dışındaki tek fark *wait/1* yerine *need/1* çağrısının kullanılmasıdır. Uygulama yönetimde hiçbir değişiklik yoktur. Sadece, akış içinde eğer *unbounded* değişkene denk gelinirse bu değişkenin beklendiği sunucuya iletilir. Talep edilen bu değişken gelene kadar *outstr/1* fonksiyonu kendini askıya alır. Yine boş küme elemanı bütün akışın kapanmasına sebep olur.

4.22. CPU Tüketimi Analizi

Hem istemci hem de sunucu tarafında yapılacak bütün işlemler giriş kuyruğundaki iletiler aracılığı ile tanımlanır. Giriş kuyruğu, hem klavye hem soketler hem de prosesin kendi iç dinamikleri ile beslenir. Giriş motoru (input engine), giriş kuyruğundan iletileri tek tek çekerek işler. Bir adet giriş kuyruğu vardır ve kuyruk üzerinde öncelik tanımlanmamıştır. Bundan dolayı kuyruğa ilk giren ileti ilk çıkan ileti olacaktır. Bir iletinin işlemesi bitene kadar diğer iletiler kuyrukta beklerler. Diğer bir deyişle tek bir *thread* mevcuttur.

call0/2 iletileri CPU zamanını en çok harcayan iletilerdir. Bu ileti işlenirken önce *head unify* yapılır, sonra *guard* işlenir. *Guard* işlendikten sonra gövde iki kısma ayrılır. Yerel çağrılar giriş kuyruğuna atılır. Uzak çağrılar ise sunucuya gönderilir. *guard* işlemleri hariç, bu işlemlerin tamamı için bir maksimum süre her zaman tanımlanabilir. Çünkü baş birleştirmesi ile gövdenin işlenmesi, giriş kuyruğuna atılması ve sunucuya gönderilmesi yazılan cümleden bağımsızdır. Cümle ne kadar uzun ve ne kadar karmaşık olursa olsun, bu işlemler belirli bir zaman aralığında tamamlanır. Aynı şekilde *call0/2* iletisi dışındaki bütün iletiler için bu mantık silsilesi yürütülebilir. Fakat *guard* çağrıları CPU zamanı harcaması bakımından muallaklık arzeder. Çünkü burada standart Prolog çağrıları yapılır ve bu çağrıların ne kadar süreceği tahmin edilemez.

Giriş kuyruğundaki *call* türevi iletilerin *guard* kısımları genel CPU tüketiminin davranışını doğrudan etkilerler. Giriş kuyruğunda, herhangi bir T anında N adet ileti mevcut ve bu N adet iletinin C adedi de *call0/2* iletisi olsun. Bütün

iletiler ve *guard kısmı hariç call0/2 iletisi*, ortalama t zamanında işlenebiliyor olsunlar. Her bir iletinin *guard* kısmı ortalama Z süre içinde işlenebiliyor olsun. $s(Z) = N \cdot t + C \cdot Z$ formülü ile verilen ortalama bir zamanda giriş kuyruğundaki bütün elemanlar işlenecektir. Görüldüğü gibi giriş kuyruğunda tüketilecek zaman Z'nin doğrusal bir fonksiyonudur. Sistem T anında $s(Z)$ süresi kadar zaman harcar. Eğer $T+S(Z)$ süresi içinde giriş kuyruğuna hiç bir ileti gelmez ise, bu süre içinde kuyruk boşalır ve proses askı durumuna geçer. Bu ideal bir durumdur.

Eğer indirgenen her *call0/2* iletisi tam bir adet yeni bir ileti doğuruyorsa kuyruk boyu sabit kalır veya azalır. En tehlikeli durum ise *call0/2* yükleminin birden fazla yerel *call0/2* üretmesidir. Bu durumda kuyruk boyu sistem kaynakları tükenene kadar uzayacaktır.

5. KÜTÜPHANE YÜKLEMLERİ

Bir yürütme modelinin pratik bir programlama dili olabilmesi için giriş/çıkış veya akış birleştirme gibi bazı özel fonksiyonlara sahip olması gereklidir. Kütüphane fonksiyonları olarak adlandırdığımız bu fonksiyonları yine Berk Prolog'un kendisi ile gerçekleştirdik. Gerçekleştirilen bu fonksiyonların birkaçı bu bölümde tanıtılmıştır.

5.1. Giriş Akışları

Paralel mantık programlamadaki en temel ihtiyaçlardan birisi kullanıcı girişlerinin *akış* şeklinde sağlanabilmesidir. Aşağıda verilen program bu amacı gerçekler. Diğer kütüphane fonksiyonlarında olduğu gibi bu fonksiyon da tek bir istemci içinde çalışır. Elde edilen sonuçların diğer istemcilere dağtılması kullanıcının sorumluluğundadır. Aşağıda verilen *instream/2* fonksiyonu daha önce anlatılan *input/2* fonksiyonunu kullanmaktadır.

```
instream(Xs, Key) ←
                input(X, Key),
                instream2(X, Xs, Key) with wait(X).

instream2([X | Xs], [X  | Ys], Key) ←
                instream2(Xs, Ys, Key).

instream2([], Ys, Key) ←
                input(X, Key),
                instream2(X, Ys, Key) with wait(X).

instream2(end_of_file, [], Key) ←
                g_assign(Key, 0),
                write(Key),
                write(' input stream closed.'),
                nl
                 |
                true.
```

Yukarıda verilen *instream/2* yüklemi [12]'de verilen input stream fonksiyonu ile hemen hemen aynıdır. Bu çok önemli kütüphane fonksiyonunun özellikleri şunlardır:

- Her *giriş akışı (input stream)* özel bir ada sahip olmak zorundadır. Adların tek olması şartı sadece aynı istemci içinde geçerlidir. Aynı ad farklı istemcilerde olabilir.

- Kullanıcı bütün girişlerini *Ad=TermList* şeklinde yapacaktır. Böylece aynı istemci içinde birden fazla giriş akışı birbirlerine karışmadan işlenebilmektedir. Örneğin,

  main ← instream(X, g1), instream(X, g2).

örneğinde aynı istemci içinde iki adet stream açılmakta kullanıcıdan iki farklı giriş beklenmektedir. Kullanıcı

g1=[a,b,c].

g2=[d, e].

g1=[f, g].

şeklinde istediği kadar ve karışık sırada giriş yapabilecektir.

- Bu girişler sırasında, istemci asla bloke olmaz. Bunu sağlamak için *input/2* yardımcı fonksiyonu kullanılır. *Key* aktif olduğu sürece input(X, Key) fonksiyonu X değişkenini sistemi bloke etmeden kullanıcıdan alır. *instream/_* yükleminde kullanılan wait(X) fonksiyonu, *input/2* yükleminden X değeri gelene kadar bütün *instream/_* yüklemini askıya alır.

- Bütün kullanıcı girişleri bir kez açıldıktan sonra *end_of_file* değeri gelen kadar açık kalır. Yukarıdaki örnekte, klavyeden, g1=end_of_file. girildiğinde bu akış (stream) kapanacaktır. Kapanan bir akışın son değeri *[ ]* olarak çağrılan yere döner.

- Bir akış kapandıktan sonra *Key* değeri iptal edilir. Stream kapandıktan sonra, kullanıcı tekrar g1=[a,b] gibi bir ifade girerse *input/2* fonksiyonu bu girişi, ikaz mesajı vererek iptal edecektir.

- Eğer kullanıcı *[]* girerse, bu giriş değeri gözardı edilir ve akış yeniden başlatılır. Bir girişi kapatmanın tek yolu end_of_file değerini girmektir.

Aşağıdaki basit örnekte, kullanıcı *fulcrum* adlı makineden sıra ile *a,b,c,d,e* değerlerini girmekte ve girilen bu değerler *phantom* adlı makinenin ekranına yazılmaktadır.

```
main ← instream(X, g1)@fulcrum,
       outstream(X)@phantom with need(X).
```

fulcrum'daki kullanıcı girişleri aşağıda verilmiştir.

```
> g1=[a, b, c].
> g1=[d, e].
> g1=end_of_file.
> g1=[f, g].
Warning: g1, undefined input stream.
```

5.2. Çıkış Akışları

outstream/1 fonksiyonu bir akışı ekranda göstermek amacı ile kullanılır. Fakat ekrana yazılacak ifade bir akış değil de tek bir ifade ise bu fonksiyonu kullanmak uygun değildir. Bu tür çıkışlar için *guard* kısmında klasik *write/1* fonksiyonu kullanılabilir. Gövde kısmında da *write/1*'in yapıtığı işin aynısını yapan *output/1* fonksiyonu tanımlanmıştır. Bu fonsiyonun açık ifadesi aşağıda verilmiştir.

```
output(X) ← write(X) | true.
```

Bu kütüphane fonksiyonunun herhangi bir özelliği yoktur. X değişkeninin *bounded* olup olmadığı kontrol edilmez. Gelen her terim olduğu gibi ekrana yazılır. Aşağıdaki örnekte aynı istemci içinde X değişkeni kullanıcıdan istenmekte ve girilen değer ekrana yazılmaktadır.

```
main ← input(X, g),
       output(X) with wait(X).
```

Aşağıdaki örnekte ise, kullanıcı girişi *fulcrum* adlı makineden yapılmakta, çıkışlar ise *phantom* adlı makinenin konsoluna gönderilmektedir.

```
main ← input(X, g)@fulcrum with share(X),
       output(X)@phantom with need(X).
```

5.3. Akışların Birleştirilmesi (Stream Merging)

Pek çok concurrent programlama dili prosesler arası iletişimi sağlamak için akışları (streams) kullanırlar. Akışlar [1]'de tanıtılan oku/yaz (read/write) işlemlerini gerçekleştirecek veri yapılarıdır. Berk Sistem de yine prosesler arası iletişim için akışları kullanır. Akışlar, yapı itibariyle listelere benzemesine rağmen, listelerden çok büyük bir farkı vardır. Akışlar, hesaplamanın herhangi bir anında parçalı olarak belirlenmiş (partially determined) veri yapılarıdır.

CCND dillerinde olması gereken en önemli özelliklerden birisi iki akışın birleştirilerek tek bir akış haline getirilmesidir (stream merging). Concurrent programlama dillerinde *stream merging* işlemleri ya hazır bir kütüphane fonksiyonu olarak verilir veya o dilin kuralları kullanılarak normal bir yüklem olarak sisteme eklenir. Berk Sistem'de ikinci yöntem kullanılmıştır.

Akışların birleştirilmesi için *merge/3* yüklemi yazılmıştır. Bu yüklem iki adet akışı birleştirmektedir. *2* adet akışı birleştirmek ile *n* adet akışı birleştirmek arasında prensipte bir farklılık yoktur. Bundan dolayı sadece iki akışın birleştirilmesi üzerinde durulmuştur. Ayrıca birleştirme için bir öncelik tanımlanmamıştır. Argümanların sırasını değiştirmekle bu özellik de *merge/3* yüklemi içine kolayca eklenebilir. Akışların birleştirilmesi özelliği Shapiro'nun raporunda [20] ayrıntılı olarak incelenmiştir.

Berk Sistem kararlı bir sistemdir. [20]'ye göre kararlı bir sistemde akışların birleştirilmesi sonlu bir bekleme (bounded waiting) ile mümkün değildir. Bundan dolayı sistemin en azından zayıf kararlı (weakly stable) veya kararsız (unstable) olması gereklidir. Kararlı bir makinede, eğer her iki akışta da aynı anda veri mevcutsa her zaman birinci akıştaki elemanlar işlenecektir. Hiçbir zaman ikinci akışa geçilemeyecektir. Fakat Berk Sistem'de uyguladığımız mimari gereği bu problem kendiliğinden ortadan kalkmaktadır. Şöyleki, her iki akışta da eleman olmasına rağmen akışları birleştirme programı *need_or* kavramı sayesinde, akışları aynı anda değil teker teker yerel belleğe çekmektedir. Aynı anda iki adet

akış olsa dahi her zaman yerel bellekte tek akış gözükecektir. Akışların isimleri *need_or* parameterlerinde ters yüz edilerek tekrar çağrıldığında bu sefer diğer akış elde edilecek ilk akışta veri olmasına rağmen, yerel makine tarafında ilk akış unbounded gibi gözükecektir.

Eğer ilk akış tarafında bir eleman bekleniyorsa, ikinci akış tarafına bir eleman geldiğinde, sistem hala ilk akışı bekleyecektir. Böylece ilk akışa bir 'ilk eleman' gelmediği sürece sistem tıkanacaktır. Bizim öne sürdüğümüz *need_or* kavramı bütün bu problemleri çözmektedir.

Berk Sistem'i için iki adet *merge* yüklemi önerilmektedir. Önerilen bu yüklemler *körleme bekleme* ve *and/or* özellikli bekleme gibi farklı özelliklere sahiptir. Önerilen ilk sistem aşağıda listelenmiştir.

```
01  merge([X | Xs], Ys, [X | Zs]) ←
02                  var(Ys)
03                   |
04                  merge(Xs, Ys, Zs) with [ need_or([Xs, Ys]), share(Zs) ].

05  merge(Xs, [Y | Ys], [Y | Zs]) ←
06                  merge(Xs, Ys, Zs) with [ need_or([Xs, Ys]), share(Zs) ].

07  merge(Xs, [], Xs) ←
08                  dummy with share(Xs).

09  merge([], Ys, Ys) ←
10                  dummy with share(Ys).
```

Berk Sistem'in yüklem çağırma mantığı gereği, *merge/3* sistemini çağrıran cümleler argümanlar üzerindeki koşulları vermek zorundadırlar. Klasik *Concurrent Prolog*'daki en büyük farklardan birisi de bu özelliktir.

Klasik bir çağrıda argüman ile ilgili *gerek* şartlar doğrudan *head* üzerinde verilebilir. Bu da çağıran yeri değil çağrılan yeri ilgilendirir. Fakat bu gerek

şartların üretilmesi Berk Sistem'de doğrudan çağıran yerin sorumluluğundadır. Buna göre, *merge/3* fonksiyonu her zaman

```
merge(Xs, Ys, Zs) with [ need_or([Xs, Ys]), share(Zs) ]
```

biçiminde çağrılır. Burada Xs veya Ys değişkeninin beklendiği sunucuya bildirilir. Klasik *merge* fonksiyonundan en büyük farkı bu özellik teşkil eder. Diğer bir deyişle hangi değişken önce gelmiş ise o değişken sunucu tarafından önce gönderilir.

need_or([Xs, Ys]) gerek şartı bu özelliği sisteme katar. *[Xs, Ys]* listesindeki argümanların sırası önemlidir. *need_or([Ys, Xs])* şeklindeki bir gerek şartta, eğer gelmiş ise önce *Ys* sonra *Xs* gönderilir. Böylece *öncelikli merger* tanımlamak için argümanlar üzerinde değişiklik yapmak yerine gerek şart üzerinde değişiklik yapılır. Bu da çok önemli bir özelliktir.

Bir an için *merge/3* yükleminin

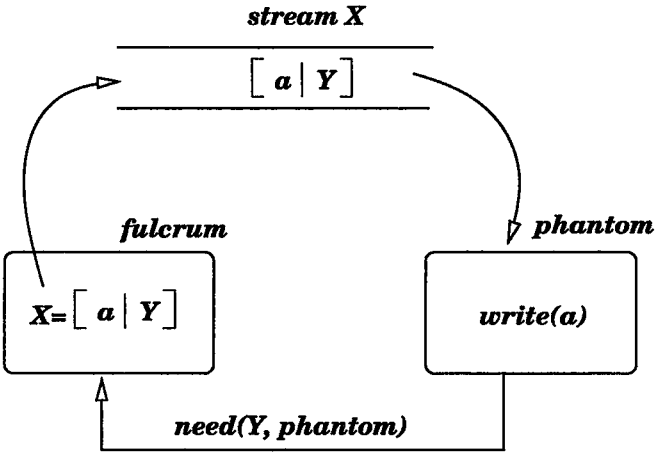
```
merge(Xs, Ys, Zs) with [ need_or([Xs, Ys]), share(Zs) ]
```

şeklinde çağrıldığını varsayalım. Eğer önce *Xs* akışı gelmiş ise *1* numaralı satır ile başlayan ilk cümle seçilir. Bunu garantilemek için *2.* satırdaki *var/1* fonksiyonu kullanılmıştır. Eğer ilk akış mevcut ise ikincisi mutlaka *unbounded* biçimindedir, *need_or/1* gerek şartı bunu garantiler. *4.* satırdaki

```
merge(Xs, Ys, Zs) with [ need_or([Xs, Ys]), share(Zs) ].
```

çağrısı tekrar *Xs* veya *Ys* şartını ortaya koyarak *merge/3* yüklemini çağırır. Akışların birleşimi olan *Zs* akışı ise *share(Zs)* şartı ile sunucuya gönderilecek ve tüketilecek olan istemciye aktarılacaktır.

Eğer ilk argüman *unbounded* ise, mutlaka ikinci argüman *Xs*, *bounded* durumdadır. Diğer bir deyişle bu argüman üzerinde bir akış mevcuttur. *5* ve *6* numaralı satırlar ikinci akışın mevcut olduğu durumlarda işleyecektir.



Şekil 5.1: Keyfi elemanlara sahip bir akışın üretilmesi

Eğer akışlardan herhangi biri kapanırsa *7.* veya *9.* satırlar ve devamındaki *dummy/0* çağrısı işleyecektir. Her iki durumda da yürütülecek olan dummy with share(Xs). cümlesi ile *Xs* veya *Ys* akışının paylaşılması veya diğer bir deyişle sunucuya aktarılması sağlanır. Buradaki uygulamada *dummy/0* fonksiyonu kullanılmıştır.

Ortak kullanılacak bir değişken ancak ve ancak *with/2* operatörü ile sunucuya aktarılabilir. Bu operatör ise bir çağrı ile kullanılmak zorundadır. Fakat bu uygulamada son iki cümlede herhangi bir çağrı olmadığı için *dummy/0* fonksiyonu kullanılmıştır. Bu fonksiyon klasik Prolog'daki *true/0* fonksiyonuna karşı gelir ve çözüm üzerinde bir etkiye sahip değildir.

Önerilen bir diğer *merge/3* yüklemi ise *need_or/1* gerek şartının *wait_or/1* ile değiştirilmesiyle elde edilir. Bu durumda beklenilen değişkenler için sunucuya herhangi bir talep yapılmaz. Körleme bir bekleme içine girilir. Üretilen değişkenlerin, bekleyen istemciye gönderilmesi değişkeni üreten istemcinin sorumluluğu altındadır. Bu özellikten dolayı önerilen bu yüklemi *bliend_merge/3* olarak adlandıracağız. Fakat programcıya çok fazla yük getirdiğinden pratikte kullanılmayacaktır.

5.4. Keyfi Elemanlı Stream Üretimi

Daha çok test amacı ile kullanılmak amacı ile keyfi elemanlı stream üretimine ihtiyaç olabilir. Aşağıdaki program keyfi elemanlara sahip, sonsuz uzunlukta stream üretmektedir. Bu örnek bir *unbounded buffer communication* örneğidir.

```
random_stream([X | Xs])  ←
                         random(X)
                         |
                         random_stream(Xs) with share(Xs).
```

Üretilen stream, *unbounded buffer* özelliğindedir. Diğer bir deyişle stream elemanları, tüketim hızından daha fazla üretilebilir, eleman üretiminin herhangi bir sınırı yoktur. Bundan dolayı üretilen bu elemanları tüketen kaynağın mutlaka, üretim hızından daha hızlı veya aynı hızda tüketmesi gereklidir. Eğer tüketim hızı üretim hızından küçük ise, belirli bir zaman sonra sistemin kaynakları tükenecek ve sistem çökecektir. Bu durumu özetleyen bir örnek aşağıda verilmiştir. Bu örnekte tüketim hızı üretimden daha yavaş olduğu için sistem çökmektedir.

```
main ←  random_stream(X)@fulcrum with share(X),
        outstr(X)@phantom with need(X).
```

Yukarıdaki örnekte *fulcrum* adlı makinede X stream'i üretilmekte ve *phantom* adlı makinede tüketilmektedir. Sistem birkaç saniye içinde çökmektedir. Yukarıdaki örnek, şekil 5.1'de ayrıca gösterilmiştir.

6. ÖRNEK UYGULAMALAR

Bu bölümde, Berk Prolog altında gerçekleştirilmiş bazı basit uygulamalar anlatılmıştır.

6.1. Sınırlandırılmış Akış (Bounded Buffer Streams)

Bir sınırlandırılımış akış (bounded buffered stream) örneği aşağıdaki programda verilmiştir. *fulcrum* tarafından üretilen akış elemanları *phantom* tarafından tüketilmektedir. Akış üzerindeki bir eleman tüketilmeden, yenisi üretilmediği için üretim sınırlandırılmaktadır. Buradaki buffer boyu 1 elemanlıktır.

```
01 main :-
02    send(X)@fulcrum with need(X),
03    receive(X)@phantom with share(X).

04 receive([X|Y]) :-
05    consumer(X) with need(X),
06    receive(Y) with [need(X), share(Y)].

07 send([X|Y]) :-
08    producer(X) with share(X),
09    send(Y) with need(Y).


10 producer(X) :-
11    g_read(no, K),
12    X is inc(K),
13    g_assign(no, X)
14    |
15    true.
```

```
16 consumer(X) :-
17   write(X),
18   nl,
19   flush_output
20   |
21   true.
```

Her bir yüklemi tek tek inceleyelim.

consumer(X) yüklemi kendine gelen X değişkenini ekrana yazarak tüketir. Bu tüketim işlemi guard kısmında yapılmıştır. X değişkeni hiç bir kısıt olmadan gövde içinde de tüketilebilirdi.

producer(X) yüklemi ise *0*'dan başlayarak birer birer sayar. Sayaç değerini X değişkeni ile çağıran yere aktarır. Diğer bir deyişle, üretim/tüketim örneği, değişkeni birer birer artırmak ve ekrana yazmak şeklinde modellenmiştir.

send([X | Y]) yüklemi ise X değişkenini üretir ve üretilen bu değişkeni

```
08 producer(X) with share(X)
```

ifadesi ile sunucuya gönderir. Bu değişkeni bekleyen tüketici derhal bunu alır ve tüketir. Daha sonra tüketici Y akışı olarak boş bir template gönderir. Bu template içinde *[Z|T]* gibi iki değişken mevcuttur.

```
09 send(Y) with need(Y).
```

ifadesi içindeki *need(Y)* şartı işte bu template'i beklemektedir. Bu template kendine gelmeden bir üretim işlemi yapmaz. Böylece tüketici, üretilecek buffer'ın template'ini verebilmektedir.

Bu örnekte beklenen Y değeri, *[Z|T]* olarak gelmektedir. Hem Z hem de T değişkeni atanmamıştır (unbounded).

send/2 yüklemi Z'yi üretecek ve benzer şekilde T değişkenini beklemeye başlayacaktır.

receive([X | Y]) yüklemine gelen stream bilgisi X ve Y değişkenleri yardımıyla işlenir. X içinde tüketilecek olan değişken mevcuttur. Bu X değişkeni

05 consumer(X) with need(X)

ifadesi ile tüketilecektir. Fakat *need(X)* fonksiyonu X değişkeninin beklendiğini sunucuya haber verir. X sunucudan gelir gelmez *consumer(X)* tarafından tüketilir. Daha sonra

06 receive(Y) with [need(X), share(Y)].

ifadesi ile *receive/1* yüklemi tekrar çağrılır. Fakat bu çağrının yapılabilmesi için önce X değişkeninin gelmiş olması garantilenmelidir. Bundan dolayı 6. satırda *need(X)* fonksiyonu kullanılmıştır. Böylece X değişkeni gelene kadar *receive/1* işlemeyecektir. X geldikten sonra *receive(Y)* çağrısı *share(Y)* ile çağrılır. Y değerine *[Z | T]* gibi bir atama yapılacak ve *share(Y)* fonksiyonu sayesinde bu *parçalı atama (partial let)* sunucuya gönderilecektir. Sunucuda bu ifadeyi *send/1* beklemektedir. *send/1* yüklemi bu parçalı atamanın head'ini tamamlar ve tekrar sunucuya gönderir. Bu sayede sınırlandırılmış akış başarılmış olur. Üretim ve tüketim hızlarının farklı olması sistemin akışını engellemez.

6.2. Bir Çakışma Örneği

Ortak değişkenler prensipte tek bir üretici tarafından üretilirler. Eğer ortak bir değişken birden fazla yerde üretiliyorsa ve paylaşılıyorsa [1] üretilen her ifadenin daha önce üretilenler ile karşılaştırılması gerekmektedir. Aşağıda verilen örnekte ortak X değişkeni iki ayrı istemci tarafından üretilmektedir. Fakat farklı değerler üretildiği için bütün sistem fail olur.

```
main :-
   (X= p(a))@fulcrum with share(X),
```

[1]Her ortak değişken paylaşılmak zorunda değildir.

```
   (X= p(b))@phantom with share(X).
```

Fakat, eğer değişkenler ortak değilse, her bir istemci aynı değişkeni farklı biçimde üretip kullanabilirler. Aslında bu da bütün prolog semantic tanımlarını değiştirmektedir. Çünkü bir değişken çözüm alanı içine bir kez değer alabilir. Fakat eğer biz bütün çözüm alanını istemcilere pay edersek, bu kural sadece istemci bazında geçerli olur, bütün çözüm ağı içinde geçerli değildir. Çözüm ağının ortak noktası olarak sunucu göz önüne alınır. Sunucu bütün çözüm ağının ortak kesişim noktasıdır. Buradaki her bir değişken tek atama kuralına uymak zorundadır. Fakat istemcide kalan ve *ortak alana düşmeyen* bir değişken ortak alandaki değerden farklı bir değer alabilir. Aşağıdaki örnekte X değişkeni her iki istemcide de farklı bir biçimde üretilmekte ve üretilen makinelerde tüketilmektedir. Üretilen değişken aynı makine içinde kaldığı için herhangibir çakışma durumu söz konusu olmayacaktır.

```
main :-
   p(X)@fulcrum,
   p(X)@phantom.

p(X)@fulcrum :-
   X= p(a),
   output(X) with wait(X).

p(X)@phantom :-
   X= p(b),
   output(X) with wait(X).
```

6.3. Klasik Multiples

Akışların birleştirilmesi ile ilgili en klasik örnek Hamming problemidir [43]. Bu problemin çözümü olan *multiples* örneği aşağıda verilmiştir. *2, 3* ve *5*'in

tekrar etmeyen çarpımları hesaplanmakta ve sonuçlar *tornado* adlı makinede listelenmektedir. [20]'de verilen çözüm ile birebir örtüşmektedir.

```
multiples :-
   stream_multiply(2, [1|X], X2)@fulcrum with share(X2),
   stream_multiply(3, [1|X], X3)@fulcrum with share(X3),
   stream_multiply(5, [1|X], X5)@fulcrum with share(X5),
   opmerge(X2, X3, X23)@phantom with [ need(X2), need(X3), share(X23) ],
   opmerge(X5, X23, X  )@mig     with [ need(X5), need(X23), share(X) ],
   outstr(X)@tornado with need(X).

multiples2 :-
   stream_multiply(2, [1|X], X2)@fulcrum with share(X2),
   stream_multiply(3, [1|X], X3)@fulcrum with share(X3),
   stream_multiply(5, [1|X], X5)@fulcrum with share(X5),
   opmerge(X2, X3, X23)@phantom with [ need(X2), need(X3), share(X23) ],
   opmerge(X5, X23, X  )@mig     with [ need(X5), need(X23), share(X) ],
   check(X, 10)@tornado with need(X),
   outstr(X)@tornado with need(X).

stream_multiply(N, [U|X], [V|Z]) :-
   V is N*U
   |
   stream_multiply(N, X, Z) with [ need(X), share(Z) ].

opmerge([U|X], [U|Y], [U|Z]) :-
   opmerge(X, Y, Z) with [ need(X), need(Y), share(Z) ].

opmerge([U|X], [V|Y], [U|Z]) :-
   U < V
   |
   opmerge(X, [V|Y], Z) with [ need(X), share(Z) ].

opmerge([U|X], [V|Y], [V|Z]) :-
   opmerge([U|X], Y, Z) with [ need(Y), share(Z) ].
```

```
check([X|_], No) :-
    X>No
    |
    freeze@server.

check([_|Y], No) :-
   check(Y, No) with need(Y).
```

Klasik multiples yönteminde çözümü durduran bir mekanizma mevcut değildir. Yukarıda verilen *mutiples2* yüklemine çözümü durduran *check/2* yüklemi eklenmiştir. Bu yüklem akış içindeki elemanlar belirli bir sayıyı aşınca bütün sistemi durdurmaktadır.

6.4. Basit Kullanıcı Giriş/Çıkışı

fulcrum adlı makineden girilen bilgiler *phantom* adlı makineden listelenmektedir.

```
main :-
   input(X, g)@fulcrum with share(X),
   output(X)@phantom with need(X).
```

6.5. Tek Değerin Giriş/Çıkışı

Herhangi bir istemciden girilen tek bir değer aynı istemcide ekrana yazılmaktadır.

```
main :-
   input(X, g),
   output(X) with wait(X).
```

6.6. Kullanıcı giriş/çıkışı

Herhangi bir istemciden girilen değerler aynı sırada yine aynı istemciden ekrana listelenir. Giriş olarak *end_of_file* değeri yazılırsa oku/yaz işlemi sona erer, akış kapatılır.

```
main :-
   input(X, g),
   outstream(X) with wait(X).
```

Farklı makinelerde giriş/çıkış akışları

fulcrum adlı makineden girilen değerler *phantom* adlı makineden ekrana listelenmektedir. *end_of_file* girişi geldiği an akış kapatılır ve program son bulur.

```
main :-
   instr(X, g)@fulcrum with share(X),
   outstr(X)@phantom with need(X).
```

6.7. Değişken Taşıma

Aşağıdaki örnekte = /2'nin kullanımı verilmiştir. Doğrudan atanan X ve Y değerleri *fulcrum* ve *phantom* adlı istemcilere aktarılacaktır.

```
main :-
   (X= [a, b])@fulcrum with share(X),
   (Y= [c, d])@phantom with share(Y).
```

6.8. Çakışma

fulcrum adlı makinede *X* değişkenine *[a, b]* değeri atanmakta, fakat aynı değişkene *phantom* adlı makinede daha farklı bir değer atanmaktadır. Paylaşılan bir değişken aynı anda iki değer alamayacağı için sistem fail olacaktır. Bu bir çakışma örneğidir.

```
main :-
   (X= [a, b])@fulcrum with share(X),
   (X= [c, d])@phantom with share(X).
```

6.9. Birleştirme

fulcrum ve *phantom* adlı makinelerden girilen değerler *mig* adlı makinede birleştirilmekte ve birleştirilen bu değerler *tornado* adlı makineden listelenmektedir. Bu örnek klasik bir *merge* uygulamasıdır.

```
main :-
   instr(X, g)@fulcrum with share(X),
   instr(Y, g)@phantom with share(Y),
   merge(X, Y, Z)@mig with [ need_or([X, Y]), share(Z) ],
   outstr(Z)@tornado with need(Z).
```

6.10. "fail" durumu

falan(a,b,c) yüklemi mevcut olmadığından bütün sistem *fail* olacaktır. Sunucu ekranına fail(falan(a,b,c)) ikazı düşecektir.

```
main :-
   write('falan ile Fail olacak.'),
   nl
```

```
   |
   falan(a,b,c).
```

6.11. fail/0

Gövde içinde *fail* kullanımı örneklenmiştir.

```
main :-
   write('fail ile Fail olacak.'),
   nl
   |
   fail.
```

6.12. if/2

Aşağıda *if/2* örneği verilmiştir. İlk argüman herzaman klasik prolog çağrılarından oluşmaktadır. Aşağıdaki örnekte *a* ve *b* sabitleri bir istemciden listelenecektir.

```
main :-
   if( true, (output(a), output(b))  ).
```

6.13. if/2

a sabiti *phantom*, *b* sabiti de *mig* adlı istemciden listelenecektir.

```
main :-
   if(true, (output(a)@phantom, output(b)@mig)
     ).
```

6.14. freeze

Herhangi bir istemci freeze@server çağrısı ile bütün çözüm ağını durdurabilir.

```
main :-
   freeze@server.
```

6.15. Unbounded Buffer

fulcrum adlı makinede *random_stream/1* yüklemi keyfi sayıda ve çok hızlı bir biçimde sayı üretir. Fakat, *phantom* adlı makinede *outstr/1* daha yavaş tüketilir. Bir kaç saniye sonra sistem boğulur. Bu bir *unbounded buffer* örneğidir.

```
main :-
   random_stream(X)@fulcrum with share(X),
   outstr(X)@phantom with need(X).
```

7. İLERİKİ ÇALIŞMALAR

Berk Prolog'un ve TCP/IP ortamında uygulanan yürütme modeli olan Berk Sistem geliştirmeye açıktır.

Berk Prolog'un son derece serbest olan semantiği daha kesin sınırlar ile tanıtılmalıdır. Dağıtık artık toplama (distributed garbage collection) sisteme eklenmelidir. Berk Prolog'un verimini, planlayıcı doğrudan etkilemektedir. Bundan dolayı hem sistemin verimli çalışması hem de kullanıcının üzerindeki yükü azaltmak amacı ile, Berk Sistem'e bir planlayıcının eklenmesi gerekmektedir. Dağıtık hesaplama için gerekli olan, *merge*, giriş, çıkış gibi yüklemler geliştirilmeli ve yenileri yazılmalıdır. Ayrıca, Berk Sistem'in kullandığı kütüphanenin tamamı sıralı Prolog ile yazılmıştır. Hızı artırmak için pek çok yüklem, doğrudan C ile yazılmalıdır. Berk Sistem için, dağıtık bellekli MIMD yürütme modeli üzerinde çalışılmıştır. Ortak bellekli modellerde sistemin çok iyi bir performans göstereceği aşikardır. Çünkü indirgeme birimleri arasında değişken akışı zaman kaybına sebep olmayacaktır. Ortak belleğe sahip bir makine için yeni bir yürütme modeli üzerinde de çalışılmalıdır.

8. SONUÇ

Bu tez çalışması ile Prolog programlama dili, paralel mantık programalamaya genişletilerek, CCND dilleri ailesine yeni bir dil kazandırılmıştır. Bu yeni dilde, kararlı bir makinenin de sonlu bir bekleme ile akışları birleştirebileceği gösterilmiştir.

Ayrıca birden fazla değişkenin şartlı olarak talep edilmesi kavramı öne sürülmüştür. Diğer klasik dillerde, önce çağrı yapılmakta daha sonra ise değişkenlerin durumuna göre proses askıya alınmaktadır. Berk Sistem'de ise bir çağrı yapılmadan evvel, çağrıya ait bütün değişkenler elde edilmekte, daha sonra çağrı yapılmaktadır. Böylece esas çağrı yapılmadan evvel, mevcut değişkenlerin durumları incelenerek, yüklemin uygun cümlesi çağrılabilmektedir. Berk Sistemi diğer sistemlerden ayıran ana özellik budur. Şartlı değişken talebi, bu özelliğin kolayca gerçekleşmesini sağlar.

Hem proseslerin hem de yüklemlerin indirgeme makinelerine kullanıcı veya planlayıcı tarafından keyfi bir biçimde dağıtabilmesi programlama açısından esneklikler getirmiştir. Bu sayede FCP veya Parlog uygulamalarında olduğu gibi değişken göçü veya ölümcül/yaşamsal kilitlenme gibi problemlerin çözülmesi zorunluluğu ortadan kaldırılmıştır.

Bunlardan belki de çok daha önemlisi sıralı Prolog programlama dili üzerinde çok az bir değişiklik yaparak paralel mantık programlama diline geçiş sağlanmıştır. Böylece paralel mantık sistemlerinin derleyicilere bağımılığı çok aza indirilmiştir.

Ayrıca Berk Prolog'un dağıtık ortamdaki yürütme modeli olan Berk Sistem tanıtılmıştır. Bu yürütme modeli, TCP/IP ortamında çalışan dağıtık bellekli bir yürütme modelidir. Bu model için, sıralı bir Prolog derleyicisi kullanılarak paralel bir derleyici geliştirlmiştir. Bu derleyici, Berk Prolog ile yazılmış pek çok test programını yürütebilmiştir. Japonların Beşinci Kuşak Bilgisayar Projesi

çerçevesinde geliştirilen KL1 sistemi bugün sadece ortak bellekli makinelerde çalışabilmektedir. Bu tür makineler ise hem çok pahalı hem de yaygın değildir. Berk Sistem ve önerilen paralel yürütme modeli Linux işletim sistemi ve TCP/IP ortamında çalıştığı için, diğer CCND dillerinden farklı olarak, donanım ve yazılım özellikleri açısından son derece yaygın bir kullanım alanına sahip olacaktır.

Berk Prolog, donanım olarak da pek çok sistemde çalışabilecektir. Çünkü yürütme modeli Linux işletim sistemi altında uygulanmıştır, ve Linux bugün donanım olarak, neredeyse dünyadaki bütün sistemleri desteklemektedir. Ayrıca sıralı Prolog derleyicilerine eklenecek ufak bir yazılım kütüphanesi (library) ile, sıralı Prolog'dan paralel mantık programlamaya geçiş kolayca yapılabilecektir. Ayrıca sıralı Prolog'un özellikleri, indirgeme elemanları bazında, Berk Prolog'a aktarılacaktır. Şu anda pratik olarak kullanımı mevcut olan KL1 ve Parlog'da bu tür bir özellik mevcut değildir.

KAYNAKLAR

[1] **Elcock, E.W.,** 1988. Absys: The first logic programming language- A Retrospective and a Commentary, *Technical Report 210,* Department of Computer Science, University of Western Ontario.

[2] **Warren, D.H.D.,** 1983. An Abstract Prolog Instruction Set, *Technical Note 309,* SRI International Artificial Intelligence Center.

[3] **Roy, P.V.,** 1994. 1983-1993: The Wonder Years of Sequential Prolog Implementation, *The Journal of Logic Programming,* **20**, 385-441.

[4] **Ait-Kaci, H.,** 1991. *Warren's Abstract Machine, A Tutorial Reconstruction,* MIT Press, Cambridge, MA.

[5] **Maier, D. and Warren, D.S.,** 1988. *Computing With Logic, Logic Programming with Prolog,* The Benjamin/Cummings Publishing Company, Inc.

[6] **Roy, P.V.,** 1984. A Prolog Compiler for the PLM, *Master's Report Plan II,* University of California, Berkley.

[7] **Kowalski, R.,** 1979. *Logic for Problem Solving,* North-Holland.

[8] **Clocksin, W.F., and Mellish, C.S.,** 1987. *Programming in Prolog,* 3rd edition, Springer Verlag, New York.

[9] **Sterling, L. and Shapiro E.,** 1989. *The Art of Prolog,* MIT Press, Cambridge Mass.

[10] **O'Keefe, R.A.,** 1994. *The Craft of Prolog,* Second Printing, MIT.

[11] **Brakto, I.,** 1986. *Prolog Programming for Artificial Intelligence,* Addison-Wesley Publishing Company.

[12] **ISO Prolog,** 2001. *Part 1, General Core,* ISO/IEC 13211-1:1995.

[13] **Pollard, G.H.,** 1981. Parallel Execution of Horn Clause Programs, *Ph.D. Thesis,* Department of Computing, Imperial College.

[14] **Khayri, A.M.A and Karlsson, R.,** 1990. The MUSE Approach to Or-Parallel Prolog, *International Journal of Parallel Programming,* **2**, 129-162.

[15] **Warren, D.H.D.,** 1987. The SRI model for or-parallel execution of prolog-abstract design and implementation issues, *In proceedings of the 1987 Symposium on Logic Programming,* 92-102, IEEE Computer Society Press.

[16] **Gupta, G. and Khayri, A.M.A. and Carlsson, M., and Hermenegildo, M.V.,** 1995. *Parallel Execution of Prolog Programs: A Survey.*

[17] **ICOT (Fifth Generation Computing), Final Conference** 1992. *Asian Technology Information Program(ATIP) Report.*

[18] **Kacsuk, P. and Wise, M.J.,** 1992. *Implementation of Distributed Prolog,* Wiley, Series In Parallel Computing.

[19] **Uchida, S.,** 1982. Toward A New Generation Computer Architecture, *ICOT Techincal Report TR-001.*

[20] **Shapiro, E.Y.,** 1983. A Subset of Concurrent Prolog and Its interpreter, *ICOT Techincal Report TR-003.*

[21] **Shapiro, E.Y.,** 1983. System programming in Concurrent Prolog, *ICOT Techincal Report TR-034.*

[22] **Shapiro, E.Y.,** 1989. The Family of Concurrent Logic Programming Languages, *ACM Computing Surveys,* **3**, 413-510.

[23] **Clark, K.L. and Gregory, S.,** 1986. Parlog: Parallel Programming in Logic, *ACM Transaction on Programming Languages and Systems,* **1**, 1-49.

[24] **Ueda, K.,** 1986. Guarded Horn Clauses, *PhD. Thesis,* University of Tokyo.

[25] **Ueda, K.,** 1986. Introduction to Guarded Horn Clauses, *ICOT Techincal Report, TR-209.*

[26] **Ueda, K.,** 1986. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, *ICOT Techincal Report, TR-208.*

[27] **Gregory, S.,** 1985. Design, Application and Implementation of a Parallel Logic, Programming Language, *PhD. Thesis,* Imperial College of Science Technology.

[28] **Ueda, K.,** 1985. Concurrent Prolog Re-Examined, *ICOT Techincal Report, TR-102.*

[29] **Roy, P.V.,** 1990. Can Logic Programming Executes as Fast as Imperative Programming ?, *PhD. Thesis,* Univeristy of California at Berkley.

[30] **Ichiyoshi, N. and Miyazaki, T. and Taki, K.,** 1987. A Distributed Implementation of Flat GHC on the Multi-PSI, *Fourth International Conference on Logic Programming,* Melbourne, Australia, 257-275, MIT Press.

[31] **Nishikawa, H. and Yokota, M. and Yamamoto, A. and Taki, K. and Uchida, S.,** 1983. The Personal Sequential Inference Machile (PSI): Its design Philosopy and Machine Architecture, *ICOT Techincal Report, TR-013.*

[32] **Nobuyuki, I. and Rokusawa, K. and Makjima, K. and Inamura, Y.,** 1988. A New External Reference Managment and Distributed Unification for KL1, *International Conference on Fifth Generation Computer Systems,* Tokyo, Japan, 904-913, OMSHA and Springer-Verlag.

[33] **Baron, U.,** 1986. A Distributed Implementation of Flat Concurrent Prolog, *Technical Report,* Weizmann Institute of Science.

[34] **Taylor, S. and Shapiro, E.,** 1987. A Parallel Implementation of Flat Concurrent Prolog, *International Journal of Parallel Programming.*

[35] **Taylor, S. and Shapiro, E.,** 1988. An Improved Parallel Execution Algorithm for Flat Concurrent Prolog, *Technical Report, CS88-09,* Weizmann Institute of Science.

[36] **Glasser, U.,** 1992. A Distributed Implementation of Flat Concurrent Prolog on Multi-transputer Environment, *Implementation of Distributed Prolog,* Peter Kacsuk and Michael J. Wise, Editor, Wiley, Series In Parallel Computing.

[37] **Akikazu, T. and Furukawa, K.,** 1986. Parallel Logic Programming Languages, *Third International Conference on Logic Programming,* 242-254, Springer-Verlag, Lecture Notes in Computer Science No.225.

[38] **Taylor, S.,** 1989. Parallel Logic Programming Techniques, Prentice-Hall.

[39] **Ackerman, W.B.,** 1982. *Data Flow Languages,* IEEE Computer **2**, 15-25.

[40] **Kahn, G. and MacQueen, D.B.,** 1997. Coroutines and networks of parallel process, In B. Gilchrist(editor), *Information Processing 77,* 993-998, North Holland.

[41] **Van Emden, M.H. and de Lucena G.J.,** 1982. Predicate logic as a programming language for parallel programming, In K. L. Clark and S. A. Tarnlund (editors), *Logic Programming,* Academic Press.

[42] **Clark, K.L. and Gregory, S.,** 1981. A Relational Language for Parallel Programming, *In Proceeding of the ACM Conference of Functional Programming, Languages and Computer Architecture, ACM.*

[43] **Dijkstra, E.W.,** 1976. *A Discpline of Programming,* Prentice-Hall.

[44] **Diaz, D.,** 2002. *GNU Prolog User Manuel, A Native Prolog Compiler with Constraint Solving over Finite Domains,* Edition 1.7.

[45] **Cohen D., Huntbach, M.M. and Ringwook G.A,** 1992. Logical Occam, *Implementation of distributed prolog,* Willey.

[46] **Wise, M.J., Jones, D.G. and Hintz, T.,** 1992. PMS-Prolog: A Distributed, Coarse -grain-parallel prolog wiht Processes, Modules and Streams, *Implementation of distributed prolog,* Willey.

ÖZGEÇMİŞ

Nazım KOÇ, 1965 yılında, İstanbul'da doğmuştur. Üsküdar Haydarpaşa Lisesi'nden 1982 senesinde mezun olmuştur. İTÜ, Matematik Mühendisliği bölümünden lisans ve İTÜ Fen Bilimleri Enstitüsü, Sistem Analizi Master programından Yüksek Lisans diplomalarını almıştır. 1992-1997 tarihleri arasında TÜBİTAK Marmara Araştırma Merkezi'nde (MAM) Osmanlıca ve Kiril harflerinin optik olarak tanınması projesinde ve EUCLID RTP 11.3 kodlu uluslararası savunma simülasyon projesinde, yapay zeka sisteminin geliştirilmesinde araştırıcı olarak çalışmıştır. Bu çalışmalar sırasında üretilen uluslararası yayınlara [1] [2] katkıda bulunmuştur. Daha sonra bir şirkette kısa bir süre fırkateyn modernleştirme projesi ve elektronik harita/yön bulma sistemlerinin tasarımı üzerine çalışmıştır. Halen özel bir şirkette yazılım mühendisi olarak, "Embedded Linux" sistemleri üzerinde çalışmaktadır.

[1] **Kocabaş, Ş., Öztemel, E., Uludağ, M., Koç, N.,** (1995). *Agents That Learn and Explain Their Actions*, Fifth Conference on Computer Generated Forces and Behavoral Representation (CGF-95), 9-11 Mayıs 1995, Orlando, Florida, A.B.D. (s. 63-68).

[2] **Kocabaş, Ş., Öztemel, E., Uludağ, M., Koç, N.,** (1996). *Design of a DIS Agent, the AISim System: A Progress Report,* Computer Generated Forces and Simulation of Behavior (CGF-96), 23-25 Temmuz 1996, Orlando, Florida. (s. 119-126).