Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-02

Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining

Jörg P. Müller

November 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Krupp-Atlas, Mannesmann-Kienzle, Philips, Sema Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining

Jörg P. Müller

DFKI-D-91-02

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITW-8902 C4).

© Deutsches Forschungszentrum für Künstliche Intelligenz 1991

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Federal Republic of Germany; an acknowledgement of the authors and individual contributors to the work; all applicable portions of this copyright notice. Copying, reproducing, or republishing for any other purpose shall require a licence with payment of fee to Deutsches Forschungszentrum für Künstliche Intelligenz.

Contents

List of Figures

Abstract

Many problems in different areas such as Operations Research, Hardware Design, and Artificial Intelligence can be regarded as constraint solving problems (CSPs). Logic programming offers a convenient way of *representing* CSPs due to its relational, declarative and nondeterministic form. Unfortunately, standard logic programming languages such as PROLOG tend to be inefficient for *solving* CSPs, since what could be called constraints in PROLOG is used only in a passive *a posteriori* manner, leading to symptoms such as late recognition of failure, unnecessary and unintelligent backtracking and multiple computation of the same solutions[1]. There have been intensive research efforts in order to remedy this. One of them, which has caught increasing attention over the past few years, is the Constraint Logic Programming approach:
By integrating a domain concept for logic variables and consistency techniques such as forward-checking or looking-ahead into PROLOG, the search space can be restricted in an *a priori* manner. Thus, a more efficient control strategy can be achieved, preserving the 'clean' dual PROLOG semantics.

In this issue, I will present a horizontal compilation approach towards a CLP system maintaining constraints whose variables are ranging over finite domains. Horizontal compilationis often referred to as *optimizing transformation techniques* in other context. A PROLOG system providing a **delay** mechanism is used in order to achieve the control behaviour described above.
The major subtasks of my work are

- Design and integration of a domain concept into logic programming, which allows direct access to and manipulation of possible values of logic variables.
- Thorough implementation of a forward-checking control strategy in SEPIA.
- Design and prototypical implementation of a looking-ahead algorithm.
- Summary of the main theoretical results underlying to domains and consistency techniques in logic programming.
- Consideration and prototypical implementation of first-fail heuristics.
- Embedding these topics into a preprocessor, which transforms FIDO programs into SEPIA programs realizing the advanced control strategies.

The general framework of this work is the FIDO lab within the ARC-TEC project, which explores several approaches towards integrating finite domain consistency techniques into logic programming.

[1] This is an observation which is true not only for CSPs but for general problems: logic programming is convenient to represent problems but its usability for solving them efficiently is restricted, since solving different types of problems require different methods. Very often these types cannot be identified from the syntactic representation only, but are connected with semantic issues.

Part I

Preliminaries

Chapter 1

Motivation

In this chapter, the motivation for my current work is described. I will outline the reasons that have led to a combination of logic programming and constraint solving. I would like to start from the logic programming "corner", showing the basic issues which made Constraint Logic Programming desirable from the logic point of view. In the second part of the chapter, I will show what logic programming has to offer w.r.t. solving constraint problems. The combination of these two aspects will lead us to the notion of constraint logic programming in a very natural way. Third and last, I'll give a short survey of the chapters following.

1.1 Logic Programming

1.1.1 The Strong Points

Since its beginning almost 20 years ago with the development [BM73, Kow74] and the first implementation [Rou75] of the language PROLOG, logic programming has developed into one of the most important tools for Artificial Intelligence. The outstanding role of PROLOG for logic programming justifies talking about PROLOG, if logic programming is actually meant[1]. The logic programming paradigm can be described by the following keywords:

- **Declarativity:** formulating knowledge in facts and rules allows the user to write *what* shall be done. *How* the task is to be performed is left to the system.
- **Relational Form:** in a mathematical sense, the knowledge items (predicates) are n-ary relations.
- **Nondeterminism:** by writing down alternatives without actually specifying a tree search strategy, nondeterminism is brought about[2].
- **Mathematical Model and Dual Semantics:** there is a well-understood underlying mathematical model for logic programs. In this context, the clear declarative semantics (least model semantics, fixpoint semantics) and the procedural semantics (which is given by SLD-resolution for PROLOG) of logic programs should be mentioned.

[1] For a strict reader, I will restrict that proposition to the logic part of PROLOG.

[2] Of course, in concrete systems like PROLOG, the order of the alternatives *is* crucial, if efficiency is taken into account.

Thus, logic programming allows problem formulation which is both elegant and natural. This also facilitates the writing of programs in such a way that they are easy to read. Furthermore, logic programming is said to shorten program development time, since it supports a *top-down* problem-solving method by dividing a goal into less complex subgoals, until the subgoals can be solved or they turn out to fail. These strong points contributed to make PROLOG a most important AI tool for both

- **Knowledge Representation:** knowledge about the world can be formulated in a first-order logic framework by facts and rules.
- **Knowledge Manipulation:** by using SLD resolution new knowledge can be derived from existing knowledge.

1.1.2 Shortcomings of Logic Programming

Unfortunately, logic programming does not have strong points only. The main negative aspects of it are

1. the unsound implementation of negation, and
2. the lack of efficient control strategies.

Unsound negation leads to wrong answers, poor control leads to inefficient problem solving results. My work stresses the control issue. First, I would like to go into more detail about what I mean by the second point of the above enumeration.
In the previous paragraph, I mentioned the capability of logic programming to allow programs to be formulated in a natural way as one of its basic advantages. However, programs written in a natural style often tend to be very inefficient. They support search strategies as *generate & test* (G&T) or standard backtracking search[3].

Generate & Test Figure 1.1 shows a program for the 8 queens problem that implements a generate & test control strategy. First, a variable assignment for all the variables is generated. In the program, that is done by creating a permutation of the values $\{1,\ldots,8\}$[4]. Second, it is tested whether the permutation generated before satisfies the safeness constraints. By G&T, there is no search space pruning at all. Constraints are used only to check whether the complete variable assignment is a solution. That fact makes G&T explore the whole search space. It performes an exhaustive search, which is very inefficient for more difficult problems as is shown by the run-time results in chapter 7.

Standard Backtracking Search A program for the 8 queens problem embodying standard backtracking search is shown in figure 1.2. The improvement compared to the G&T algorithm is the following: each time a value is assigned to a variable, it is tested whether that value is consistent with the values of the variables assigned before the current variable. If this is not the case, backtracking occurs, going back to the most recent choice point and trying another value there. That way, an obvious failure can be detected before values have been given to all

[3] We certainly can achieve more sophisticated control mechanisms in PROLOG (i.e. forward-checking), but that will lead to programs neither natural nor easy to understand.

[4] Note, that due to that representation, the constraint which excludes two queens from standing in the same row is made implicit.

```
\* permute(List, Permlist) succeeds if Permlist is a permutation of List *\
    eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
        permute([1,2,3,4,5,6,7,8], [X1,X2,X3,X4,X5,X6,X7,X8]),
        safe([X1,X2,X3,X4,X5,X6,X7,X8]).

    safe([]).
    safe([H|T]) :-
        no_attack(H,T),
        safe(T).

    no_attack(X, Y) :-
        no_attack(X, Y, 1).

    no_attack(X, [],_).
    no_attack(X, [H|T], N) :-
        X =\= H + N,
        X =\= H - N,
        N1 is N + 1,
        no_attack(X, T, N1).
```

Figure 1.1: G&T Program for the n Queens Problem

variables. Standard backtracking achieves an *a posteriori* search space pruning, which makes it essentially superior to generate & test algorithms. However, it has some serious disadvantages, basically induced by the backtracking mechanism. These will be described in the following.

```
eight_queens([X1, X2, X3, X4, X5, X6, X7, X8]) :-
    queens_aux([X1,X2,X3,X4,X5,X6,X7,X8], [], [1,2,3,4,5,6,7,8]).

queens_aux([], Placed, []).
queens_aux([H|T], Placed, Values) :-
    delete(H, Values, Newvalues),
    no_attack(H, Placed),
    queens_aux(T, [H|Placed], Newvalues).

\* no_attack / 2 and no_attack / 3 are the same as in the G&T program! *\
```

Figure 1.2: Standard Backtracking Program for the n Queens Problem

Backtracking Backtracking [CM81] has the advantage of being a simple search strategy which can be easily implemented. Unfortunately, backtracking-directed control mechanisms suffer from a "disease" which can be characterized by the following symptoms:

- Late detection of failures.
- Continuous rediscovery of identical partial solutions.
- Unintelligent selection of choice points, i.e. the true culprit of a failure is often detected very late, involving a lot of redundant work beforehand.
- Useless node generations in the search tree.
- Recovering instead of avoiding of failure. Backtracking starts only after a failure has occured.

From the computational point of view, backtracking is known to be of exponential complexity in the worst case. Thus, many interesting problems cannot be solved within a reasonable time using standard backtracking search.

1.1.3 Remedies

Starting from Kowalski's [Kow79] famous equation

$$\textbf{algorithm} = \textbf{logic} + \textbf{control},$$

we can summarize that the shortcomings of today's general logic programming PROLOG systems arise within the control area. That is a very serious problem, since huge search spaces are typical for many AI problems. Handling these search spaces efficiently, however, can only be done if a sophisticated control mechanism is available which avoids unnecessary, exhaustive search. That is why there have been intensive search efforts aimed at improving the control mechanisms of PROLOG. An important branch of research in that area was e.g. coroutining for PROLOG, which basically allows G&T programs to perform standard backtracking search (see section 6.3.1). Another interesting aspect was finding intelligent backtracking mechanisms in order to optimize choice point selection [SS77, Bru78, Bru81]. The main criticism about using coroutining mechanisms is that coroutined PROLOG does not remedy the negative symptoms induced by standard backtracking search. Although it is true that intelligent backtracking can basically improve the efficiency of standard backtracking, it mainly *recovers* its shortcomings, thus only remedies the symptoms, but not the disease. In my opinion, it would be better to *avoid* failure a priori, whenever that can be done.
Thus, from the perspective of logic programming, an active a priori reduction of the search space is desirable. That means not to wait until a failure has occured and react to it, but to avoid producing failures by eliminating inconsistent variable values.
For this purpose, consistency techniques such as *forward-checking* or *looking-ahead* are good options. They not only guarantee the consistency between the current variable assignments with assignments made before, but also use information about the currently known variable values (or value sets) in order to eliminate inconsistent values from the domains of variables that have not been instantiated yet.
The paradigm of constraint logic programming (CLP) [JL87] embodies this idea in an outstanding manner. Its principles will be introduced in chapter 2.
In this work, I will present the design and the implementation of a CLP-like control mechanism, which allows to make use of uses forward-checking in order to solve efficiently constraint problems in logic programming. This will offer a way to overcome the above mentioned shortcomings of logic programming languages w.r.t. to control.

1.2 Constraint Solving Problems

A lot of interesting problems can be regarded as instances of constraint solving problems (CSPs). Such problems are e.g. graph colouring, graph isomorphism, scene and edge labeling, logical puzzles or boolean satisfiability [van89a]. Many real-world problems such as scheduling or warehouse-location problems can be transformed according to one of these representation classes. In the following, I would like to point out what logic programming has to offer with respect to solving CSPs, and how logic programming can benefit from methods used for solving CSPs.

What makes logic programming especially well-suited for stating constraint problems, are the relational form it provides, and its nondeterminism.

- **Relational form:** since constraints are nothing but relations between objects symbolized as variables, they can be formulated naturally and conveniently in logic programs.
- **Nondeterminism** liberates the programmer from doing explicit tree search and allows declarative formulation of problems.

Therefore, logic programming seems appropriate for *stating* constraints, and so for stating e.g. discrete combinatorical problems. Unfortunately, standard logic programming does not support efficient methods for solving CSPs. Therefore, the logic programming scheme should be extended by more efficient control mechanisms, as they exist for constraint solving.

Constraint Solving Techniques Constraint solving is a well-understood problem solving method which has been subject to intensive research. There are several standard algorithms for constraint solving, e.g.

- Generate & Test.
- Standard Backtracking.
- Forward-Checking.
- Looking-Ahead.
- Specialized methods for solving linear equations and disequations, such as the Gaussian and Simplex methods.

As we have seen, G&T and standard backtracking are naturally integrated into logic programming. From the point of view of constraint solving, it is interesting to integrate the more efficient techniques such as forward-checking and looking-ahead into an extended logic programming scheme. Whereas specialized constraint solving techniques are not taken into consideration in FIDO, a complete integration of forward-checking is realized. This is described in chapter 6.2.3. Looking-ahead is implemented only in an exemplary way and in a modified form (see section 6.2.4). The theoretical foundations of these techniques are presented in chapter 3.

1.3 Overview

In the following, the overall structure of this work will be outlined.

Chapter 2: Constraint Logic Programming In chapter 2, the overall framework of constraint logic programming is presented. The first part of the chapter contains an outline of the principles of CLP. In the second part, some important systems are described in short. Emphasis is laid on the comparison of the capabilities of these systems to what FIDO is supposed to perform.

Chapter 3: The Theoretical Framework In this chapter, the theoretical foundations concerning the integration of finite domains and consistency techniques into logic programming are summarized and the fundamental definitions are given.

Chapter 4: The Role of FIDO-II within the FIDO Lab Here, the location of this work within the FIDO project is described. The different subprojects of FIDO are described and preliminary results are reported.

Chapter 5: A Domain Concept for FIDO-II In this chapter, a domain concept for logic programming is presented. After describing domain concepts in a more general way, the concrete implementation in FIDO-II is presented. A further paragraph deals with some problems caused by the domain variable representation in FIDO-II.

Chapter 6: The Integration of Control Here, the realization of the second major issue of this work is outlined, which is the integration of advanced control strategies in PROLOG. First, the notion of constraints in FIDO-II is introduced. Second, I present the implementation of the important consistency techniques by SEPIA delay declarations. Third, the heart of FIDO-II, the preprocessor performing the horizontal source-to-source transformation is explained. The chapter ends with some remarks on a programming methodology in FIDO-II.

Chapter 7: Applications Chapter 6 demonstrates the scope of FIDO-II on several exemplary applications taken from different problem classes such as logical puzzles, graph colouring problems and scheduling. The improvement in efficiency, which is partially drastic compared to using standard logic control mechanisms is shown by some computational results. The performance is also compared to some other CLP systems revealing the limitations of the approach.

Chapter 8: Summary and Outlook In the final chapter, the main results of the work are summarized. Some problems and limitations are shown, and an outlook is given as regards further research efforts within FIDO.

Chapter 2

Constraint Logic Programming

2.1 The Principles of CLP

The drawbacks of logic programming outlined in chapter 1 have led to intensive research efforts aimed at improving the control facilities of logic programming languages. One approach, to which increasing attention has been paid over the past few years, is contraint logic programming (CLP). The main idea of CLP is to combine the strong points of logic programming, which are its declarativity, its simple and clear semantics based on a well-understood mathematical model and its nondeterminism, with the efficiency of constraint solving techniques such as forward-checking and looking-ahead. Thus, it becomes possible to formulate a large class of combinatorical problems in a natural and elegant way *and* to solve them efficiently. The crucial point is that the use of consistency techniques provides the ability of pruning the search space in an active, *a priori* manner. Values that are known to be inconsistent with the current variable states can be excluded from further consideration.
In the following, I would like to point out what actually has to be done in order to bring about the combination between logic programming and constraint solving that characterizes CLP.

2.1.1 The CLP Scheme

The notion of *constraint logic programming* has been coined by [JLM86, JL87]. It was designed to generalize the semantics of the logic programming scheme w.r.t. a particular equational theory. Thus, the CLP scheme **CLP**($\mathcal{T}$) was born. Its syntax is a definite clause syntax.
It is remarkable that the underlying theory is left unspecified. By instantiating it with a special equational theory[1], instances of the CLP scheme can be created, inducing a class of constraints relevant for that theory. Examples of instances can be found in section 2.2.
Thus, by constraint logic programming, a class of programming languages is defined whose instances share the same essential semantic properties. For this purpose, some new semantic concepts had to be introduced.
Meanwhile, in CLP, the original equational theories have been extended by special-purpose (non-equational) theories.

2.1.2 The Structure of a CLP system

The general structure of a constraint logic programming system reflects its purpose: the combination of logic programming and constraint solving. Figure 2.1 shows the organization of a

[1] An important requirement for this theory is its unification completeness.

CLP system. Basically, it consists of two components, an inference machine doing the logic

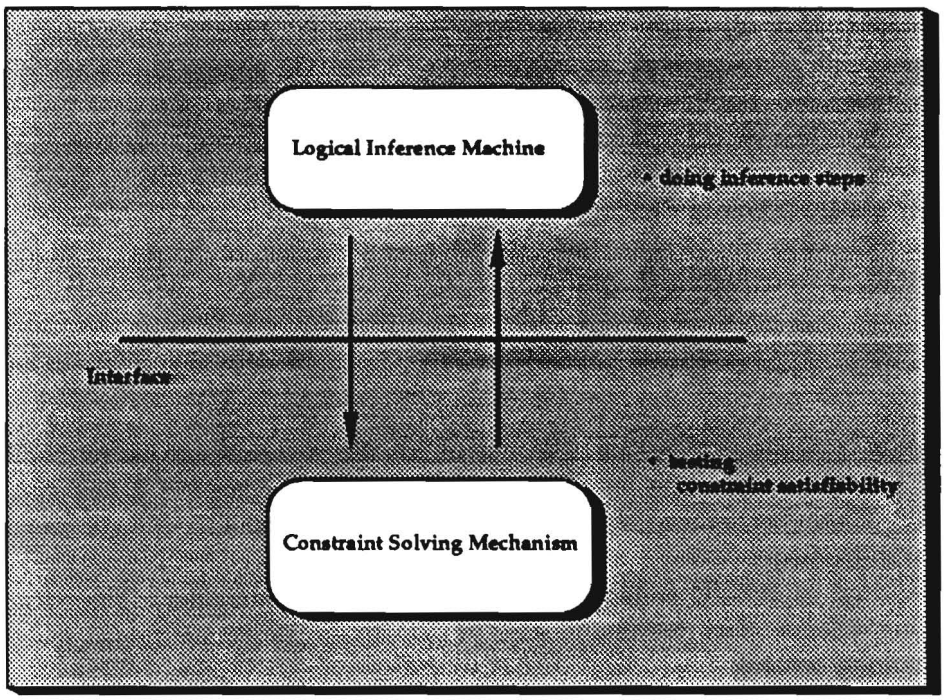

Figure 2.1: The General Structure of a CLP System

part, and an incremental constraint solver, which can be considered as a decision procedure for a class of constraints. The two modules communicate by an interface. The inference machine recognizes constraints and passes them to the constraint solver. The latter one incrementally creates a constraint net and tries to solve it, reporting the results back to the inference machine[2].

2.1.3 Finite Domain Consistency Techniques

In the approach underlying my work, techniques for solving constraints over finite domains are to be examined. In the following, some considerations will be made about what is needed to do this:

Finite Domains: The restriction to finite domains is convenient for us, since it allows an explicit representation of the set of values an object (symbolized by a variable) can have. What we need is a *domain concept* for logic variables which enables us to restrict the domain of a variable in an active way. In chapter 3, the theoretical foundations of domains in logic programming are summarized. In chapter 5, its realization in FIDO-II is described.

[2]In real systems, this way of job-sharing can be somehow varying, e.g. some simple constraints can be directly solved by the inference engine.

Consistency techniques: Here, we have to define the tools to be used in order to achieve an advanced control, realizing an active a priori use of constraints. These tools will be *forward-checking* and *looking-ahead*:
Forward-checking can be applied in order to restrict the domain of a variable X appearing in a constraint C, if all other variables of C are ground. Then, C can be regarded as a unary predicate, and, since constraints must be decidable, the remaining values for X satisfying C can be computed. Looking-ahead (which is neglected a bit in this work for reasons explained later on) can be applied to a constraint, even if more than one of its arguments are unbound. Thus, looking-ahead leads to an earlier restriction of the search space. However, it is far more expensive than forward-checking.
Formal definitions of the consistency techniques are given in chapter 3, their implementation in FIDO-II is described in chapter 6.

2.1.4 Properties of CLP

In chapter 1, we have seen that the control strategies supported by PROLOG imply a poor control behaviour, resulting in search strategies such as generate & test and standard backtracking. These mechanisms drastically restrict the applicability of logic programs to complex real-world problems.
Backtracking, especially, suffers from some pathological maladies. The constraint logic programming scheme offers reasonable ways to remedy the above-mentioned shortcomings of logic programming control:

- Constraints are used in an active manner, positively influencing program control.
- The search space is kept small, since early pruning is done.
- Less choice points are generated. Thus, fixing the real culprits of a failure is a much easier job than it is in logic programming.
- Since a priori pruning is performed, inconsistencies are detected earlier. Less failures are produced and unnecessary backtracking is avoided.
- The explicitness of domains allows the use of first-fail heuristics (see section 6.5).
- A large class of problems can be formulated in an elegant and natural way by using constraints and consistency techniques.
- The semantic background of logic programming can be essentially preserved in CLP. This has been shown by [JL87, van89a].

2.1.5 Summary

Here, I would like to summarize those aspects of a CLP system which are of special importance for this work.

1. A domain concept for logic variables is introduced.
2. Consistency techniques are included in the PROLOG computation model.
3. The PROLOG inference engine is coupled with a constraint solver[3].

[3] As we will see, in FIDO-II, these two components will be nested.

4. The CLP scheme is instantiated with an equational theory over finite domains.
5. Constraints shall be used in an active, a priori manner.
6. The negative effects of standard backtracking search can be avoided.
7. The clear semantics of logic programming is basically preserved (see chapter 3).
8. The scope of logic programming languages can be extended to $\mathcal{NP}$ complete discrete combinatorical problems.

2.2 The State of the Art

In this paragraph, a brief overview is given of the most important existing CLP systems and their features. I would like to start with CLP($\mathcal{R}$), since it has been developed by the team of Joxan Jaffar and Jean-Louis Lassez, which can claim to "have invented constraint logic programming". The second important system is the CHIP system which has been developed by the team of Pascal van Hentenryck. It has been a good[4] benchmark for FIDO, since it can handle finite domains, too. Apart from these, I would like to mention PROLOG-III, TRILOGY and the recent work of van Hentenryck, the advanced finite domain CLP system CC(FD).

2.2.1 CLP($\mathcal{R}$)

CLP($\mathcal{R}$) [JL87, JM87] can be looked upon as a special instance of the CLP scheme. The underlying domain are the real numbers, the constraints are linear equations and inequalities. The methods used for constraint solving are the following:
First, there is a delay-mechanism for non-linear constraints. These are suspended until they become linear. Linear equations are solved by Gaussian elimination whereas a Simplex algorithm [Dan63] is used for linear inequalities. The main applications for CLP($\mathcal{R}$) are electrical engineering problems, option trading and other problems requiring reasoning over the reals.

2.2.2 CHIP

CHIP [DvHS+89] stands for **C**onstraint **H**andling **I**n **P**ROLOG. Domains in CHIP can be rational numbers, boolean terms and finite domains. As regards domains, CHIP substantially differs from systems such as CLP($\mathcal{R}$) or PROLOG-III, since there is no restricted set of built-in constraints. In CHIP, the user can define logic programs as constraints. This gives the CHIP approach much more flexibility. The only restriction which stems from the constraint solver is the linearity condition for constraints.
The methods used for constraint solving in CHIP are a Simplex algorithm both for linear equations and disequations. For finite domains, forward-checking, looking-ahead and partial looking-ahead algorithms are available. The main application areas of CHIP are operations research problems, such as scheduling, graph colouring etc.

[4] Well, actually, it has been *too* good!

2.2.3 PROLOG-III

Prolog-III [Col87a] has been developed subsequent to PROLOG-II by A. Colmerauer. The domains supported by it are rational terms (which are represented as infinite rational trees), strings and boolean terms. The latter can be considered as a very special type of trees. Thus, although it is represented differently, PROLOG-III could be considered as "CLP($\mathcal{Q}$)" and could be integrated as an instance of the constraint logic programming scheme. Constraints in PROLOG-III can be equality constraints on boolean terms, linear equations and disequations and inequalities on rational terms. Besides, PROLOG-III provides advanced numerical and list processing features. The solving methods in PROLOG-III are the following:

- for boolean equations, a saturation method combined with SL resolution is used.
- linear constraints are solved by a Simplex algorithm.

The main application areas for PROLOG-III are circuit analysis, banking calculation, operations research problems and, due to the complete boolean algebra it provides, reasoning rules for expert systems.

2.2.4 TRILOGY

Basically, TRILOGY [Vod88] could be viewed as "CLP($\mathcal{Z}$)" although it is presented differently. The language is cut to integer programming. The underlying theory are the integers with the addition operator. The only constraints are numerical built-in constraints. The constraint solver is a decision procedure for integers.
As far as I know, the methods used in TRILOGY are quite expensive and thus, not well-suited for solving complex real-world problems. So far its only applications have been logical puzzles.

2.2.5 CC(FD)

CC(FD) [van91] is the recent work of van Hentenryck. It has been presented on the occasion of a talk at the PDK workshop 1991 in Kaiserslautern [RB91]. Basically, it presents a further development of the finite domain handling capabilities in CHIP. Especially, CC(FD) provides features handling drawbacks of actual constraint programming languages. A common problem arises e.g. by disjunctive constraints, which appear in disjunctive scheduling problems. CC(FD) copes with these by extracting information from the disjunctions in order to use them for pruning, if this is possible. Choices can be avoided in the case of so called deterministic disjunctions. Other features of CC(FD) are the active use of cardinality and projection constraints. Besides, it is possible to express universal quantification. Thus, a more efficient handling of nondeterminism in CLP and to a better control behaviour can be achieved. However, information about practical results of CC(FD) has not yet been available to me.

2.2.6 Relations to FIDO

Basically, we can divide the systems presented above into two groups. The first group contains CLP($\mathcal{R}$), PROLOG-III and TRILOGY. These systems can be characterized by the fact that the constraint solver is a black box which can handle only a set of built-in constraints. The second group is represented by the CHIP system. It does not use a complete constraint solver, since

most of the intended applications are $\mathcal{NP}$-complete problems. Therefore, using a complete solver would be too inefficient. Furthermore, the user can specify how constraints shall be used, and the system is not restricted to a set of built-in constraints, but logic programs can be considered as constraints.
FIDO can be classified as belonging to the second group, since consistency techniques can be applied to arbitrary constraints, even if it is not completely implemented in the current prototype version of FIDO-II, e.g. as regards looking-ahead. Besides, the user can define how constraints are to be solved. Thus, the functionality of FIDO can be considered as being similar to CHIP.
However, the philosophy behind the FIDO project is quite different. Up to now, it has not been our intention to build *the* newest, best and fastest constraint programming language[5]. We are studying the usability of some approaches aiming at a realization and, since each approach has strong points and drawbacks, the idea is to combine the approaches by making use of the strong points of each approach. This is especially true for the cooperation between FIDO-II, where a horizontal compilation is performed, and FIDO-III, which is currently worked on, and which implements a vertical compilation approach of finite domain constraints. Combining these two paradigms seems to be a promising idea.
For a more detailed description of the FIDO project and the relation of FIDO-II to it, see chapter 4.

[5] Maybe tomorrow!

Chapter 3

Theoretical Framework

3.1 Motivation

One of the very strong points of logic programming is its mighty underlying mathematical model. The clear declarative and procedural semantics allows an elegant and simple programming style. But this comes at a price in a twofold way:

- Semantics is defined within the restrictive context of the Herbrand universe.
- Only syntactically equivalent terms are unifiable.

That ties logic programming into a hermetic framework. Therefore, many recent PROLOG systems incorporate some equality theory in order to master the case of semantically equivalent, but syntactically different terms. Note also that arithmetic is usually dealt with in an ad-hoc way in PROLOG. The problem with these approaches is that they not always preserve the logic base, although this is a crucial requirement on extensions to logic programming languages. PROLOG-II e.g. was defined as a rewriting system in the domain of infinite rational trees and *not* as a logic programming language. In [JMSY90], I found the nice citation:

"Forsaking the logic in PROLOG in order to remove some limitations of the language is like throwing out the baby with the bath water."

Generalizing the logic programming paradigm to the constraint logic programming scheme turns out to be a solution of that dilemma. The generalization made it possible to describe systems as PROLOG-II and PROLOG-III again as instances of the scheme.
A reference for the theory of logic programs is [Llo84]. A more general approach to logic and logic calculi can be found in [Ric78]. The foundations of CLP have been worked out by Jaffar and Lassez [JL87, JM87] and by van Hentenryck [van89a] for the case of finite domains. In this chapter, I would like to outline the most important results of that research, as far as FIDO-II is concerned. Especially, I will skip the results concerning uncountable domains (CLP($\mathcal{R}$)), since these are not relevant for FIDO-II. In the following, I will summarize how a domain concept can be integrated into the logic programming scheme. In further paragraphs, the inference rules underlying to the consistency techniques used in FIDO will be defined and some of their properties will be described.

3.2 Domains in Logic Programming

In [van89a], van Hentenryck shows how a domain concept can be formally embedded into a logic programming language. The declarative and procedural semantics of first-order languages with domain variables are given.
First, I would like to define what a domain is.

Definition 1 *A domain d is a non-empty set of constants.*

In the following I concentrate on a finite set $\mathcal{R}$ of domains which can be looked upon as the domains used by a program. Especially, $d \in \mathcal{R} \Rightarrow e \in \mathcal{R}$ for $e \subset d, e \neq \emptyset$.

3.2.1 Declarative Semantics

A declarative semantics of a first-order language Γ with domain-variables can be defined according to the following steps:

1. Enhance the alphabet of Γ by a set of domain variables for each $d \in \mathcal{R}$. x_d denotes a domain variable of the domain d.
2. Define terms, definite programs and goals as usual.
3. Give the declarative semantics of a logic program with domain variables by extending the model theoretic semantics of first-order logic:
   - Define the interpretation $\mathcal{I}$ of a domain d as the assignment of a subset d' of the universe D of the usual interpretation.
   - extend the interpretation of formulas $\exists x_d$.P and $\forall x_d$.P as follows:
     $\mathcal{I}(\exists x_d.\mathrm{P})$ = true iff there exists c $\in$ d' with $\mathcal{I}^{x_d,c}(\mathrm{P})$ = true.
     $\mathcal{I}(\forall x_d.\mathrm{P})$ = true iff $\mathcal{I}^{x_d,c}(\mathrm{P})$ = true for all c $\in$ d'.
4. Define the notions *logical consequence*, *Herbrand interpretations* and *models* as usual.
5. Define *substitution*, *correct substitution* and *correct answer substitution* as usual, however, providing a special treatment for domain variables:
   a domain variable x_d can be substituted by
   - a value c of its domain, or by
   - a domain variable y_e, ranging over a domain $e \subset d$.

Note in that context that domains are basically interpreted as unary predicates. We write $c \in d$ instead of $d(c)$.

3.2.2 Procedural Semantics

The procedural semantics in PROLOG is given "by the way it works", i.e. by SLD resolution based on the unification algorithm [Rob65]. In the case of a logic programming language with domain variables, procedural semantics can be described basically the same way. The difference to SLD resolution in PROLOG is that unification is modified in order to handle domain variables. First, I should like to introduce the notion of *unifier* and *most general unifier* of two expressions. It can be defined as usual:

Definition 2 (unifier, mgu) *Be t_1, t_2 expressions of a first-order language with domain variables. σ, ρ, τ be substitutions.*

a) t_1 and t_2 are unifiable with unifier σ iff $\sigma(t_1) = \sigma(t_2)$.

b) Be t_1, t_2 unifiable with unifier σ. σ is called the most general unifier of t_1, t_2 iff for each unifier ρ of t_1, t_2, $\rho = \tau \circ \sigma$, i.e. $\sigma \leq \rho$ for each unifier ρ of t_1 and t_2.

The mgu can be extended to sets $E = \{t_1, \ldots, t_n\}$ of terms as usual. Now, I would like to give the redefinition of the unification algorithm for domain variables. The following three additional cases have to be taken into consideration.

- unification of a normal variable with a domain variable. In that case, the normal variable is bound to the domain variable.
- unification of a domain variable with a constant. If the constant is member of the domain of the variable, unification succeeds and the domain variable is bound to the constant. Otherwise, unification fails.
- unification of two domain variables. If the intersection of the two domains is not empty, unification succeeds and binds both variables to a new one ranging over that intersection. If the intersection is singleton, both variables are bound to the remaining constant. I the intersection is the empty set, unification fails.

According to the above requirements, a unification algorithm for a first-order language can be defined as shown in figure 3.1. The algorithm receives as input a set of expressions. Its output is the mgu for E (if it exists) or *fail*. The notion of *disagreement set* in the algorithm is understood as usual. Theorem 1 asserts termination, soundness and completeness of the algorithm for finite sets of expressions.

Theorem 1 *For the domain variable unification algorithm 3.1 the following holds:*

- *a) The algorithm always terminates.*
- *b) Let E be a finite set of expressions. If E is unifiable, the unification algorithm gives the mgu σ for E. If E is not unifiable, the unification algorithm reports that.*

The proofs of a) (termination) and b) (which is a generalization of Robinson's unification theorem have been given in [van89a]. □

In the following, SLD resolution based on the domain variable unification algorithm will be referred to as SLDD resolution.

As a conclusion, we can say that domain variables can be embedded into logic programming, enforcing some extensions but preserving the main results such as the declarative and procedural semantics of the language. By extending unification, an active handling of the equality constraint is provided. Yet, in order to maintain other constraints that way, additional inference rules have to be defined, embodying the consistency techniques used in this work.

```
k := 0; σ_k := ε;
while Flag = true do
 {
  if singleton(σ_k)
  then
   { return σ_k(E); STOP }
  else
   {
    compute_disagreement_set(σ_k(E), D_k);
    if member(v, D_k) and member(t, D_k)
                      and not occurs_in(v, t)
    then \* unification of simple variable and term *\
     { σ_{k+1} := σ_k[v←t]; k := k + 1 }
    else
     if member(v_d, D_k) and is_domvar(v_d) and member(a, D_k)
                         and is_constant(a) and member(a, d)
     then  \* unification domain variable - constant *\
      { σ_{k+1} := σ_k[v_d←a]; k := k + 1 }
     else
      if  member(v_d1, D_k)and is_domvar(v_d1) and member(v_d2, D_k)
                          and is_domvar(v_d2, D_k) and d_2⊂d_1
      then  \* unification of two domain variables over the same domain *\

       { σ_{k+1} := σ_k[v_d1←v_d2]; k := k + 1 }
      else
       if  member(v_d1, D_k) and is_domvar(v_d1) and member(v_d2, D_k)
                            and is_domvar(v_d2, D_k) and d_1∩d_2 ≠ ∅
       then  \* General unification of two domain variables *\
        {e := d_1∩d_2; σ_{k+1} := σ_k[v_d1←w_e, v_d2←w_e]; k := k + 1 }
       else
         Flag := false;
  }
  RETURN("not unifiable!");
  STOP
}
```

Figure 3.1: A Domain Variable Unification Algorithm

3.3 Consistency Techniques in Logic Programming

3.3.1 Constraints

First, I want to define what a constraint is. A constraint is characterized by the fact that the inference rules defined in the next section can be applied to it. Therefore, it must fulfil the following conditions:

Definition 3 *An n-ary predicate p is a constraint iff for any ground terms t_1, ..., t_n one of the following is true:*

- *$p(t_1,\ldots,t_n)$ has a successful refutation, or*

- *$p(t_1,\ldots,t_n)$ has only finitely failed derivations*

In other words, a predicate p is a constraint, if all its ground instances either succeed or finitely fail.

3.3.2 The Forward-Checking Inference Rule

In this section, the forward-checking inference rule (FCIR) is presented. The FCIR can be applied to a constraint if only one of the variables appearing in the constraint, say X, is left uninstantiated. The domain of that variable can then be restricted by deleting those domain values from Dom_X that don't satisfy the constraint[1]. First, I would like to define formally when the FCIR can be applied to a predicate.

Definition 4 *Be $p(t_1,\ldots,t_n)$ an atom. We say that $p(t_1,\ldots,t_n)$ is forward-checkable, if*

- *p is a constraint*
- *there exists only one t_i, $1 \leq i \leq n$, that is a domain variable, all others being ground.*

t_i is often called the forward variable. Now, I will define the FCIR:

Definition 5 (The FCIR) *Be P a program, $G_i = ?- A_1,\ldots,A_k,\ldots,A_m$ a goal and σ_{i+1} a substitution. G_{i+1} is derived by the FCIR from G_i, P, σ_{i+1}, if the following holds:*

1. *A_k is forward-checkable, x_d be the forward variable inside A_k.*
2. *The new domain e is defined as*
   $e = \{a \in d | P \models A_k\{x_d \leftarrow a\}\} \neq \emptyset$.
3. *Then, the new substitution σ_{i+1} is defined as*
   $\sigma_{i+1} = \{x_d \leftarrow c\}$, if $e = \{c\}$, or $\{x_d \leftarrow y_e\}$, where y_e is a new domain variable, otherwise.
4. *$G_{i+1} = ?- \sigma_{i+1}(A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_m)$ a goal and σ_{i+1} a substitution.*

Note that, due to the definition of constraints, the new domain e in point 2 can be computed easily (e.g. by using SLDD resolution). With the help of the FCIR, an a priori pruning can be achieved, because values are actively eliminated from the domain, i.e. are not considered again. Another important point is that the forward variable becomes instantiated if its domain is singleton (see point 3 of definition 4).
The FCIR has some nice properties. The central ones will be summarized in the following.

Theorem 2 (Soundness of the FCIR) *Be P a program, G_i be the goal $?- A_1,\ldots,A_k,\ldots,A_m$. A_k be forward-checkable, x_d be the forward variable. Be $d = \{a_1,\ldots,a_n,b_1,\ldots,b_k\}$, d, $e \neq \emptyset$. Let the goal G_{i+1} be obtained from G_i as $G_{i+1} = ?- \sigma_{i+1}(A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_m)$. Then G_i is a logical consequence of P iff G_{i+1} is a logical consequence of P. We also write $G_{i+1} \equiv_P G_i$.*

[1] Here, the constraint is regarded as a unary predicate.

The proof to this theorem can be found in [van89a]. The theorem gives expression to the fact that by a derivation step using the FCIR, no wrong results will be achieved, thus the soundness of the derivation is preserved. In the following, I will denote first-order resolution extended by the forward-checking inference rule as SLDFC resolution.
The soundness result can be expressed alternatively as follows: If $B_1, \ldots, B_n$ is a proof sequence in the SLDFC calculus, then

$$B_1, \ldots, B_i \vdash_{SLDFC} B_{i+1} \Rightarrow B_1, \ldots, B_i \models B_{i+1} \quad \text{for } 1 \leq i \leq n.$$

A result which makes the FCIR particularly interesting from a computational point of view is its completeness. In [van89a], the completeness result has been proved for a procedure that

- uses the FCIR for forward-checkable predicates and
- uses normal derivation, otherwise.

I will denote such a proof procedure as SLDFC resolution.

Theorem 3 (Completeness of FCIR) *P be a logic program, G be a goal. If an SLDD refutation of $P \cup \{G\}$ exists, then there also exists an SLDFC refutation of $P \cup \{G\}$. Moreover, if σ is the answer substitution from the SLDD-refutation of $P \cup \{G\}$, and ρ is the answer substitution from the SLDFC-refutation of $P \cup \{G\}$, then $\rho \leq \sigma$.*

For the proof of theorem 3, I would like to refer again to [van89a].
Informally, the completeness of the FCIR means that each provable goal is also reached using forward-checking inference steps in the way specified above. The reason for the completeness of the FCIR is its very strong application condition. As long as a goal is not forward-checkable, only normal derivation will be applied to it. Moreover, in a computation sequence, the FCIR can be applied at most once to *one* goal, resulting in failure or further restriction of the domain of the forward-variable.

3.3.3 The Looking-Ahead Inference Rule

In this chapter, the looking-ahead inference rule (LAIR) will be defined. By that rule, constraints can be used, even if more than one variable is left uninstantiated. Thus, the LAIR generally leads to an earlier pruning of the search space than the FCIR. First, I will define the applicability conditions for the LAIR:

Definition 6 *An atom $p(t_1, \ldots, t_n)$ is lookahead-checkable if*

- *p is a constraint and*
- *There exists at least one t_i that is a domain-variable. All other t_j are either ground or domain variables.*

Definition 7 (The LAIR) *Be P a program, $G_i = ?- A_1, \ldots, A_k, \ldots, A_m$ a goal and σ_{i+1} a substitution. G_{i+1} is derived by the LAIR from G_i, P, σ_{i+1} if the following holds:*

1. *A_k is lookahead-checkable, $x_1, \ldots, x_n$ are the lookahead variables of A_k.*

2. *For each* $x_j, e_j = \{v_j \in d_j | \exists v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}, v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ *with* $\sigma(A_k), \sigma = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$ *is a logical consequence of* $P\} \neq \emptyset$.

3. y_j *is the constant* c *if* $e_j = \{c\}$ *or a new variable which ranges over* e_j, *otherwise.*

4. $\sigma_{i+1} = \{x_1 \leftarrow y_1, \ldots, x_n \leftarrow y_n\}$.

5. G_{i+1} *is either* $? - \sigma_{i+1}(A_1, \ldots, A_{k-1}, A_{k+1}, \ldots, A_m)$ *if at most one* y_i *is a domain variable, or* $? - \sigma_{i+1}(A_1, \ldots, A_m)$, *otherwise.*

Since, for this work, looking-ahead is not as important as forward-checking, I would like to shorten a bit the theoretical considerations and refer to [van89a] for a more detailed discussion of them.
It should be mentioned, however, that for the LAIR, there is **no completeness** result whereas its **soundness** can be proved. The reason for that fact is that looking-ahead only does a pruning of the search space, but never makes choices. That basically means that we need to combine normal derivation and the LAIR in order to yield a sound *and* complete proof procedure. Particularly, we will have to use normal derivation also for lookahead-checkable goals in order to achieve global consistent solutions. An example for this can be found in figure 3.2, which describes an instance of the n-colouring of a complete n-graph for $n = 3$.

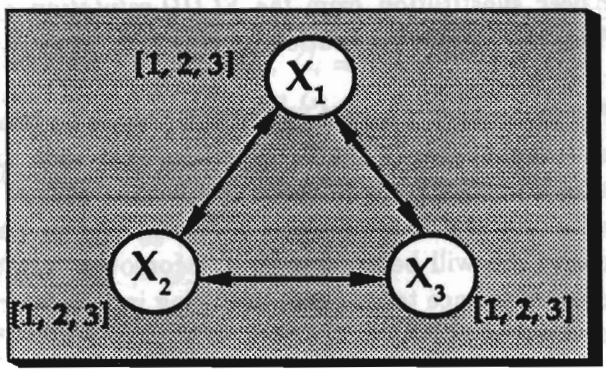

Figure 3.2: 3-Colouring of a Complete 3-Graph

An inference rule which uses looking-ahead for lookahead-checkable goals and normal SLDD resolution for other goals will never find a solution to the problem. Since the LAIR detects local consistency for all domain values, it will never eliminate any values from one of the domains. Moreover, all constraints remain always lookahead-checkable, so that normal derivation steps will never be made. In order to find a solution, we will have to make choices, e.g. we could instantiate X_1 to 1 and see what happens. That seems natural because for solving some problems, it is necessary to make choices.
Practical consequences of that issue for FIDO-II are described in section 6.2.5. An algorithm embodying the use of the LAIR for prepruning and the use of normal derivation (combined with the FCIR) can be found in the next paragraph 3.3.4.

3.3.4 Weak Looking-Ahead(WLA)

The weak looking-ahead strategy combines the use of LAIR and FCIR. That technique has been informally proposed in [dSPRB90] as "first-order looking-ahead". I prefer to call it weak

looking-ahead because I think that this name is more appropriate for expressing what the underlying algorithm really does. The basic issue about WLA is that each constraint can be selected by the looking-ahead part *at most one time.* After that, only the FCIR (or normal inference) can be applied to it. That idea is caught by the following definitions.

Definition 8 *An atom $p(t_1, \ldots, t_n)$ is called WLA-checkable if*

1. *p is a constraint.*
2. - *$p(t_1, \ldots, t_n)$ is lookahead-checkable and has not been selected by the LAIR, yet, or*
   - *$p(t_1, \ldots, t_n)$ is forward-checkable.*

Definition 9 (WLA) *Be P a program, $G_i = ?- A_1, \ldots, A_k, \ldots, A_m$ a goal and σ_{i+1} a substitution. G_{i+1} is derived by the WLA from G_i, P, σ_{i+1} if the following holds:*

1. *A_k is WLA-checkable, with $x_1, \ldots, x_n$ being the WLA variables in A_k.*
2. - *If A_k is lookahead-checkable and the LAIR has not been applied to A_k in the actual proof, then goto 3.*
   - *Otherwise set $\sigma_i' := \sigma_i$ and goto 5.*
3. *For each $x_j, e_j = \{v_j \in d_j | \exists v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}, v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ with $\sigma_i'(A_k), \sigma_i' = \{x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n\}$ is a logical consequence of $P\} \neq \emptyset$.*
4. *y_j is the constant c if $e_j = \{c\}$, or a new variable which ranges over e_j, otherwise.*
5. *$G_i' = ?- \sigma_i'(A_1, \ldots, A_m)$.*
6. *G_{i+1} can be derived from G_i', P, σ_{i+1} using SLDFC resolution.*

Properties of Weak Looking Ahead The main point of the above definition is point 6, which uses SLDFC resolution, whose soundness and completeness have been proved, in order to finish the proof after some prepruning has been done using the LAIR in a definite way. Thus, if we want to proof soundness and completeness of the WLA, we basically have to check the LAIR part. Since that part is involved at most one time for each goal, and since this happens as early as possible (due to point 2 of the definition), the negative aspects of the LAIR such as its incompleteness can be avoided. For the proof of the soundness of WLA we need the following lemma, whose proof can be found in [van89a]

Lemma 1 *Let P be a program, $d = \{a_1, \ldots, a_k\}, e = \{a_2, \ldots, a_k\}$, A be an atom containing x_d as variable, Q be a conjunction of atoms and $x_d, x_1, \ldots, x_n$ be the variables appearing in A and Q. Assume that $\exists x_1 \ldots \exists x_n . A[x_d \leftarrow a_1]$ is not a logical consequence of P. Then $\exists x_d \exists x_1 \ldots \exists x_n . (A \wedge Q)$ is a logical consequence of P iff $\exists z_e \exists x_1 \ldots \exists x_n . (A \wedge Q)[x_d \leftarrow z_e]$ is a logical consequence of P.*

Theorem 4 (Soundness of WLA) *Be P a program, G_i be the goal $? - A_1, \ldots, A_k, \ldots, A_m$. A_k be WLA-checkable. Let the goal G_{i+1} be derived by WLA from G_i, P, σ_{i+1} as $G_{i+1} = ? - \sigma_{i+1}(A_1, \ldots, A_{k-1}, A_{k+1}, \ldots, A_m)$. G_i is a logical consequence of P iff G_{i+1} is a logical consequence of P, denoted by $G_{i+1} \equiv_P G_i$.*

Proof: The proof will be performed in two steps.

1. show that $G_i \equiv_P G'_i$.
2. show that $G'_i \equiv_P G_{i+1}$.

Having shown that, with 1), 2) and the transitivity of $\equiv_P$ (which can be proved easily) follows $G_{i+1} \equiv_P G_i$.
ad 1)
Case 1: G'_i is derived from G_i by selecting A_k from G_i with A_k is not lookahead-checkable or A_k has not been selected by the LAIR in former inference steps. Then $G_i \equiv G'_i \Rightarrow G_i \equiv_P G'_i$.
Case 2: G'_i is obtained from G_i by selecting an A_k which is lookahead-checkable and has not yet been selected by the LAIR. Then, for each WLA-variable x_j, apply lemma 1 for all values which are included in d_j, but which are not included in e_j. Doing that delivers that G_i is a logical consequence of P *iff* $\sigma'_i(G_i)$ is a logical consequence of P. Since $G'_i = \sigma'_i(G_i)$, we have G_i is a logical consequence of P *iff* G'_i is a logical consequence of P. That is equivalent to $G_i \equiv_P G'_i$.
ad 2)
Since we only use SLDFC resolution steps in order to derive G_{i+1} from G'_i (see point 6 of the definition), it is obvious that $G'_i \equiv_P G_{i+1}$ because of the soundness of the FCIR and of SLDD-resolution. □
A very nice property of WLA is its completeness. This means that we can define a complete proof procedure using weak looking ahead. This is done in the following

Definition 10 (SLDW-resolution) *A first-order resolution proof procedure is called SLDW-resolution, if it uses*

- *weak looking-ahead for WLA-checkable goals and*
- *normal SLDD-derivation for other goals*

We can prove completeness of such a proof procedure by making use of the completeness of the FCIR (see theorem 3), showing that by applying the LAIR at most once to each goal, no solutions are lost. For this, we need the following lemma.

Lemma 2 *Be G a goal, P a program, Ψ an SLDW refutation of P∪{G}. If P is not lookahead-checkable or the LAIR has been applied to G in Ψ before, then the following holds:*

a) G is WLA-checkable iff G is forward-checkable.

b) G_i can be derived by the WLA from P∪{G} with answer substitution σ iff G_i can be derived by the FCIR from P∪{G} with answer substitution σ.

c) There exists an SLDW refutation Ψ of P∪{G} iff there exists an SLDFC refutation Φ of P∪{G}. Moreover, $\Psi = \Phi$.

Proof:
ad **a)** By the definition of "WLA-checkable", if G is not lookahead-checkable or the LAIR has been applied to G before, and if G is WLA-checkable, then G must be forward-checkable and vice versa.

ad **b)** Assume G_i can be derived from $P \cup \{G\}$ by the WLA with substitution σ. By our hypowork, G is not lookahead-checkable or the LAIR has been applied to G in a former inference step. Then, by definition 9, point 2, a WLA derivation step corresponds exactly to a derivation step using the FCIR for forward-checkable predicates and using normal derivation for others. It follows directly that G_i can be derived by the FCIR from $P \cup \{G\}$ with answer substitution σ. The other direction of the equivalence can be shown analogously.
ad **c)** Using b), we can do the proof by induction on the number of derivation steps of the SLDW refutation Ψ for $P \cup \{G\}$, showing that each application of a WLA derivation is also an instance of an FCIR derivation step, showing that the resulting SLDFC refutation Φ of $P \cup \{G\}$ is equal to Ψ. □
Now we can show the completeness of weak looking-ahead. It is expressed by the following theorem.

Theorem 5 (Completeness of WLA) *P be a logic program, G be a goal. If there exists an SLDD refutation of $P \cup \{G\}$, then there also exists an SLDW refutation of $P \cup \{G\}$. Moreover, if σ is the answer substitution from the SLDD-refutation of $P \cup \{G\}$, and ρ is the answer substitution from the SLDW-refutation of $P \cup \{G\}$, then $\rho \leq \sigma$.*

Proof:
Assume G is WLA-checkable, and there exists an SLDD refutation of $P \cup \{G\}$ with answer substitution σ.
Case 1: G is not lookahead-checkable or the LAIR has been applied to G earlier during the derivation. Then, by the completeness of SLDFC resolution (theorem 3) there exists an SLDFC refutation Ψ of $P \cup \{G\}$ with answer substitution ρ and $\rho \leq \sigma$. By our hypowork and by lemma 2.c, Ψ is also a SLDW refutation. Thus, there exists a SLDW refutation Ψ of $P \cup \{G\}$ with substitution ρ and $\rho \leq \sigma$.

Case 2: G is lookahead-checkable and the LAIR has not yet been applied to G. Then, due to the definition of WLA, the next derivation step is performed using the LAIR on G. Be $x_1, \ldots, x_n$ with domains $d_1, \ldots, d_n$ the lookahead variables in G, $\delta = \{x_1 \leftarrow z_1, \ldots, x_n \leftarrow z_n\}$ be the substitution resulting from applying the LAIR to G. We can restrict ourselves to the cases where the z_j are either constants c_j or domain variables z_{ej} with $e_j \subset d_j$. The answer substitution σ of the SLDD refutation of $P \cup \{G\}$ can be considered as a suitable restriction of δ to the variables of G. Thus, it follows that there exists an SLDD refutation of $P \cup \{\delta(G)\}$ with answer substitution θ, which is the same as the SLDD refutation of $P \cup \{G\}$ except the original goal and the unifier.
Now, due to the completeness of the FCIR (theorem 3), there also exists an SLDFC refutation of $P \cup \{\delta(G)\}$ with answer substitution τ and $\tau \leq \theta$. Since the LAIR is never again used on $\delta(G)$, we can apply lemma 2.c to derive that there exists an SLDW refutation of $P \cup \{\delta(G)\}$ with substitution τ. It follows that there exists an SLDW refutation of $P \cup \{G\}$ with answer substitution $\rho = \delta \circ \tau$, and $\rho \leq \sigma$. □

WLA uses looking-ahead only one time for each predicate call, thus yielding a prepruning of the search space. After that, control is left to SLDFC resolution, which has been shown to be complete. Thus, a goal G_i with a substitution σ_i is transformed by WLA into a goal G_i' with a substitution $\sigma_i' \leq \sigma_i$. Since

- WLA only reduces the search space and

- WLA always uses normal inference resp. SLDFC resolution after the looking-ahead prepruning,

for a search procedure based on SLDW-resolution the completeness result can be confirmed.

3.3.5 Consistency

The kind of consistency provided by techniques as looking-ahead is *k-consistency*. As we will see later, in FIDO-II we need an additional condition in order to assert global consistency instead of k-consistency. It will be the condition that choices are made in order to detect inconsistencies. That means, variables have to be instantiated. Otherwise, e.g. the global inconsistency of the constraint net shown in figure 3.3 cannot be detected. At the first glance,

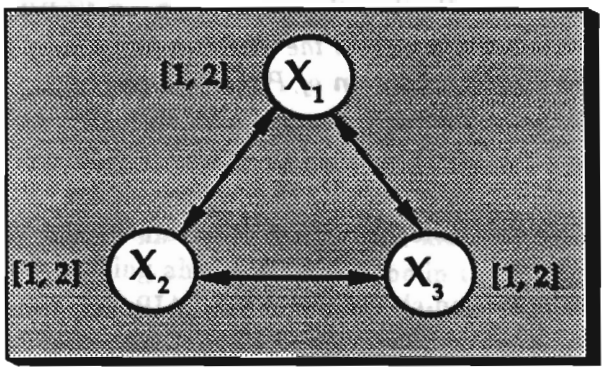

Figure 3.3: A Globally Inconsistent Constraint Net

that seems incompatible with the correctness and soundness of the FCIR. Note, however, that none of the three constraints shown in figure 3.3 ever becomes forward-checkable unless one of their arguments is given a value, i.e. the FCIR could not be applied to the above constraint net. Thus, the putative incompatiblity turns out not to be one. It is a general problem with the consistency techniques ensuring k-consistency described in this work that global consistency is only guaranteed if all variables appearing in the constraints have been assigned a value. The consequences of that fact for FIDO-II will be pointed out in section 6.2.5.

Part II

FIDO-II: Concepts and Implementation

Chapter 4

The Role of FIDO-II within the FIDO Lab

In the FIDO lab, which has started in autumn 1990, certain approaches towards an integration of finite domain consistency techniques into logic programming are to be examined. By that, experience concerning the strong points and shortcomings of single approaches shall be gathered. The ultimate goal, which just turns out to be visible at the horizon (but how far away can this be ?!), is something like a CLP system working over hierarchically sorted domains. An approach to a hierarchical CLP system is described in [BMMW89]. This goal itself can be considered as an intermediate result of FIDO, because when we started from scratch almost a year ago, the direction to go was not so clear. But let me stop here, referring to chapter 8 for a more detailed description of the results.
Up to now, FIDO is divided into three approaches, which are very loosely coupled. These are

- the meta-interpretation approach (**FIDO-I**).
- the horizontal compilation approach (**FIDO-II**), which is decribed in this work.
- the vertical compilation approach (**FIDO-III**).

In the following, I should like to give a short presentation of the two other approaches, FIDO-I and FIDO-III.

4.1 The Meta-interpretation Approach

This approach ([Sch91]) is essentially based upon [Hol90b]. SEPIA is enhanced by forward-checking, using extended unification and meta-interpretation. A most important notion in this context is the one of meta-terms and meta-variables:
Applying the definition of *domain variable unification* as specified in chapter 3 to domain variables in PROLOG implies a problem, since it is actually not possible to reassign values to logic variables. The way chosen here to escape from that problem is to make use of open data structures together with a backtracking mechanism. Meta-terms as introduced in [Hol90a] are such open data structures. A variable can be bound to a meta-term consisting of a reference to the variable itself (i.e. a new variable which is bound to a unique value *iff* the variable itself is to be bound to that value), furthermore of an open data structure whose last non-ground element incorporates the actual state of the variable (respectively of its domain), and of a list of references to all constraints the variable appears in.
Domain variable unification and constraint propagation will be handled by meta-interpretation.

4.1.1 Domain Variable Unification

If two meta-variables X and Y are to be unified, a meta-unifying routine picks up the actual variable states from the meta-terms X_{meta} and Y_{meta} representing X resp. Y. The intersection of the two domains is computed and added to the meta-terms as new current variable state. If this causes the domains to become singleton, the singleton value is assigned to the variables. The unification between a domain variable and a normal variable or a constant is handled similarly. For a more detailed discussion of this, especially as regards domain representation, see [Sch91, Hol90b].

4.1.2 Achieving Forward-Checking Control

Corresponding to the definition of the forward-checking inference rule (see chapter 3), a *forward* call to a constraint has to be delayed if the constraint is not *forward-checkable*, i. e. not sufficiently instantiated. Later on, if all constraint variables up to one (which must be a domain variable) have received a value, the constraint will be resumed and executed. This behaviour is obtained by storing constraints that are not forward-checkable in a constraint list. At the same time, with each variable submitted to a constraint, a reference to this constraint is stored. Thus, every time a variable is instantiated (or released again on backtracking), the constraints concerning this variable are reconsidered and executed or resuspended[1]. The process of reconsidering constraints is an interesting topic itself: doing it in an unsmart manner induces a lot of redundant work by testing all constraint variables, even if only the value of a single variable has actually changed. A more detailed discussion of this problem and a proposal for its solution can be found in [Hei91].

4.1.3 Results

Using FIDO-I, some of the most known benchmark applications could be formulated and solved, such as N queens, 5 houses and various puzzles. However, reading the short system description above should have caused a light feeling of unease, for the implementation of domain variable unification and control via meta-interpretation let us expect a poor efficiency behaviour of FIDO-I, compared to systems as those described e.g. in [Col87b, DvHS+88, JL87, Vod88].
Unfortunately (or should I rather say: fortunately?) this was exactly what happened. The system turned out to be relatively slow. Computing a solution of the N queens problem for example was a hard trial of the user's patience for $N > 12$. For exact performance data, I refer to the applications in chapter 7.
But, nevertheless, it has been a first step, providing us with essential experience needed for our further proceeding.

4.1.4 What We Have Actually Learned from FIDO-I

The main point responsible for the relatively poor performance of the system has doubtlessly been the high overhead needed on one hand for handling control, on the other for domain variable unification and domain access. Surely, the performance of the system could be improved up to a certain grade by a more sophisticated representation and implementation. But, essentially, the problem is a general one. It is a refresh of the well known fact that, the higher the level of abstraction is, the poorer performance tends to be.

[1] if execution is not possible

Thus, the conclusion we had to draw was to shift the main system elements described above to deeper system layers. Our hope was that a deeper integration of domain handling and control would pay off - and up to now, this hope seems to verify.
Therefore, we decided to start two further, more elaborate approaches: A horizontal compilation approach (FIDO-II), which is described in this work, and a vertical compilation approach (FIDO-III) [Hei91].

4.2 The Vertical Compilation Approach

The results delivered by FIDO-I and FIDO-II brought to daylight that constraint systems based upon an existing PROLOG and written in PROLOG can hardly compete with deeply integrated systems such as e.g. [JL87, DvHS+88] for more complex real-world problems. Therefore, an integration of finite domain consistency techniques into the WAM [War83] seems to be a promising way, keeping the advantages of PROLOG such as declarativity, nondeterminism, dual semantics and relationality on one hand, and avoiding too much overhead on the other. Thus, the main idea of FIDO-III is to compile finite domain constraints vertically into a WAM architecture, extending the basic WAM data structures and using a freeze-like control scheme [Car87].
The two major ingredients to be implemented here are a mechanism for the advanced control strategies (forward-checking and looking-ahead) and an extension of the WAM unification routine which can cope with domain unification. This leads to a set of new WAM instructions specific to domain variables.
This topic is just worked on in our group. A brief sketch of the current state of the work can be found in [MHM91].

Chapter 5

A Domain Concept for FIDO-II

Together with chapter 6, this chapter can be looked upon as the heart of the master work. Based upon the foundations laid up to now, I will give an overview of FIDO-II. In this chapter, the realization of a domain concept is described, which we need in order to introduce variables ranging over finite domains. After having a general view at conceptual and implementation issues concerning domains in logic programming, the actual realization of a domain concept in FIDO-II is described. Finally, an alternative possibility of representation is shortly discussed, which is based on a new feature of SEPIA PROLOG, the *meta-terms*.

5.1 A General View on a Domain Concept

In this subchapter, the first important concept of FIDO-II shall be presented. We will see how domains can be integrated into PROLOG, enabling us to process variables ranging over finite domains. First, we will discuss the requirements on a domain concept and the design alternatives resulting from these. Thereafter, the concept finally implemented in FIDO-II is presented.

5.1.1 Motivation

PROLOG logical variables range over the Herbrand universe, which is the domain of the ground instances of terms. The only means to handle (or manipulate) these variables in standard PROLOG is by *unification*. However, unification can be looked upon as a very special instance of constraint solving, since it involves substitution, and since each substitution can be viewed as a destructive step for solving the (syntactical) equality constraint. Particularly, unification provides no means to handle the domains in an a priori manner. The search strategies supported are standard backtracking and generate & test as described in chapter 2.
What we want is an a priori use of *constraints* (as a generalization of the unification notion), thus yielding an early and effective restriction of the search space. Therefore, the logic programming scheme shall be enhanced by a domain concept:

- To each variable, a finite domain is attached which can be specified by the programmer. The possibility of accessing domain elements and restricting the domains shall be provided.
- In a second step, which is pointed out in the next subchapter (6), we will show how these domains are processed: *unification* will be generalized to *solving constraints*. An advanced

N	# Domain Accesses[2]
4	24
8	473
12	1323
16	493
32	1635
48	3904

Table 5.1: Number of Domain Accesses for the N Queens Problem

control strategy will be introduced, which is capable to access domains directly and manipulate them. Thus, inconsistent values can be eliminated early from the domains, leading to a soon detection of failure without doing a lot of backtracking beforehand.

The idea of using domains for variables in logic programming is not new. It can be found in mathematical logic [End72], in automatic theorem proving [MM77a, MM77b, Coh83], in unification theory [Wal84a, Wal84b] and in systems for knowledge representation [BFL83]. The LOGIN system [AkN86] has introduced this idea for logic programming. What is new in the CLP approach is that the domains contribute to an *a priori* pruning instead of being used statically during unification. This is why the use of domains described here differs substantially from the approaches above.

5.1.2 Requirements on a Domain Concept

Introducing domains for logic variables obviously implies a certain overhead. Each time a variable is involved in an action, i.e. it appears as an argument of a constraint, the domain has to be accessed in order to lookup for values satisfying the constraint, simultaneously eliminating inconsistent values from the domain. After each manipulation of a domain, it must be tested whether it has become singleton. In this case, the remaining value will be assigned to the variable.
It is important to become aware of the fact that domain access can turn out to be a bottleneck, whose efficient realization greatly determines the performance of a finite domain CLP system, since this kind of variable manipulation is a very fundamental operation here (whose importance is well comparable to PROLOG unification). Table 5.1 shows the number of domain accesses for a forward-checking version of finding the first solution to the N queens problem for several N, thus trying to give the feel for how often domains are accessed even in relatively small programs[1].

Thus, the main requirements on a domain concept can be summarized as follows:

- Fast, if possible direct access to single domain elements.
- Efficient representation of "valid" and "invalid" domain elements.
- Structural knowledge about domains should be used if available in order to speed up search.
- Fast initialization (construction) of domains, if possible at compile time.

[1]The statistics was created using the SEPIA statistics facility.

- Support to the domain variable intersection operation.
- Easy undoing of operations on the domains[3].
- Preserving the clear and dual PROLOG semantics.
- Simple and declarative syntax.
- Hiding internal details from the point of view of the user as much as possible.

In a more general framework and for future extensions of the system, additional requirements could be thought of, such as

- Dynamic modifiability and extendibility of domains.
- Support of the representation of hierarchical taxonomic knowledge, e.g. coupling with a KL-ONE like representation language.
- Connection to a database system.

Let us now have a look at the alternatives which be chosen in order to satisfy the requirements listed above, before we will describe how domains are actually integrated in FIDO-II. I will not say much about the three last points I called additional requirements. These points are not taken into consideration in this work (neither they are in the FIDO lab). They could be, however, worth some reflection for the future work.

5.1.3 Domain Representations

The most important decision on the design, which influences all other points mentioned above, consists of choosing an appropriate domain representation. This especially affects the possibilities of an efficient access to domain elements. In the following, I will specify what a good solution looks like in order to being able to classify the quality of the representation chosen in FIDO-II. For that, some alternatives are to be described w.r.t. the way they support the requirements listed above.

5.1.3.1 Domain Representation as General Lists

Representing a domain simply as a list of its elements is an alternative which can be achieved very simply. The language PROLOG offers features supporting elegant list processing, which are even extended if special PROLOG hardware is available. Thus, efficient sequential access to domain elements is guaranteed.
But, in our context, we often need the capability of directly accessing domain elements. Consider e.g. a forward-checking use of the $\neq/2$ constraint. Assume there is a constraint of the form $X \neq Y$, X and Y being domain variables.
Assume that Y has become ground and instantiated with the value 2. X be a domain variable ranging over the domain $\{1,2,3\}$. Since Y is ground, the constraint has become forward-checkable (see chapter 3) and it can be executed. Here, it has to be tested whether the domain (in the example the domain of X) contains a special value (the value 2). Obviously, a sequential search through the domains happens to be quite inefficient, since, in general, finding an

[3]This can be described by the nice word "backtrackability".

element in a domain containing N elements requires an average of $N/2$ tests. In worst case, the whole domain has to be scanned. Especially for bigger domains, direct access (or at least binary search) would lead to a remarkable improvement in efficiency.

5.1.3.2 Domain Representation as Ordered Lists

This representation, which is a special case of the general list representation, allows the use of binary search methods, having to test $\ln N$ elements in the worst case in order to find the one desired (or to fail). Besides, it can be very useful for handling constraints relating to intervals, such as the $>$ and $<$ constraints. The disadvantages of this representation, however, are the overhead induced by creating and maintaining the ordered lists and the lack of flexibility of the concept w.r.t. dynamic changes within domains. Moreover, a domain can be physically ordered only by *one* criterion, which also contributes to the inflexibility of the concept. Using PROLOG as representation language in this context, even an order on the domain elements does not provide direct access, because PROLOG lists can only be processed in a sequential manner.

5.1.3.3 Using Trees for Representing Domains

An adequate means to guarantee efficient access to domain elements is the use of tree structures. In PROLOG, trees can be represented recursively as uninterpreted functors whose arguments form the subtrees [CM81, Bra86, SS86]. The use of sort and search trees for data storage and retrieval is well understood and has been a research topic for a long time in the field of database systems[Hä87][4]. The tree structures used most frequently are

- Binary trees
- Digital trees
- B and B^* trees.

The use of binary trees is described in [Wir75, Knu75, Nie74]. A more theoretic approach can be found in [Meh75]. Digital trees are discussed in [Mor68, HB78]. The use of B and B^* trees[BM72, Wed74], which enforce good flexibility w.r.t. semantic search criteria by overlaying several access paths over a data set, seems to pay off only for big domains (as found in database systems), which, in particular, cannot be kept in the main storage and have to be held on secondary storage media. The access (paging) is managed by a sophisticated buffer administration.
In most CLP applications, i.e. discrete combinatorical problems, domain sizes are relatively small, i.e. they will hardly exceed some 100 elements, so that the overhead caused by these methods will probably not pay off. This is the reason why we can restrict ourselves here to considering sorted binary trees. Using these tree structures in order to speed up search might surely be a promising approach, especially if domains are static, i.e. the trees representing domains can be built once and for all during compile time. The main problems with binary trees is on the one hand their low fan-out, on the other it is quite difficult to keep the trees balanced [Knu75]. Especially if domains are created dynamically, this causes remarkable cost. That is the reason why binary trees are commonly not used in database systems.

[4] In database systems, the leaves of these trees normally consist of pointers to whole pages, not to single data entries, since in this area, one has to face the necessity of secondary storage access.

5.1.3.4 Using Hashing Methods

The use of hash techniques in order to facilitate direct access is commonly considered as a good approach w.r.t. efficiency and implementation effort [ML75, Fag79, Cha81]. Doubtlessly, if only direct access is needed, hashing is about to be nearly optimal. But, there are two shortcomings of using hashing algorithms for data storage and retrieval:

- The problem of finding a *good* hashing function, which guarantees an equi-distribution of domain elements into hash classes for arbitrary domains. There is no general solution for this problem up to now, although, as shown in [H87], there exist some standard methods which gain good results in most cases.
- For each domain element, the hash adress must be computed every time it is accessed. This makes hashing inefficient once sequential access to data elements is required[5].

5.1.4 Classifying Valid and Invalid Domain Values

An important feature of FIDO-II (and so of all CLP systems) is its ability to prune the search space in an a priori manner by eliminating values from the domains. Sometimes, if assumptions made before turn out to be wrong (i.e. they fail), the elimination of values which were based upon these assumptions has to be revised and the elements deleted before have to be restored. Therefore, we must find an appropriate representation of validity and invalidity of data elements, which must be

- easy to check

and which has to

- allow easy revising of changes made beforehand.

Obtaining that by explicitly deleting data elements is certainly no good idea. Especially if we use sophisticated data structures such as ordered lists or even trees for domains, the process of revising a change, i.e. re-inserting an element in its old place, can cause an unbearable overhead which can result in rebalancing the complete tree structure.
Another idea, which is proposed in [dSPRB90], is to make use of the PROLOG backtracking facilities by adding a *valid/invalid* flag to each domain element. This can be achieved by a variable which is instantiated once the element is deleted. The main advantages of this technique are:

- Since value and flag are stored together, once you have found the domain value, you get the flag for free, without any further search.
- The process of revising modifications can simply be left to the PROLOG backtracking mechanism.

5.2 Bringing about Domains in FIDO-II

Starting from the considerations made in paragraph 5.1, I will now point out how domains are integrated in the implementation of FIDO-II.

[5]This can be the case in our field, i.e. for the domain variable singleton test.

5.2.1 Domain Variables in FIDO-II

In this paragraph, I describe the representation of domain variables ranging over finite domains. To each logic variable that is bound to be used as a domain variable, a finite domain must be attached. The realization of this link can be carried out in several ways. In particular, a *global* or a *local* approach can be chosen.

- **Global link:** Here we could think of a global table with an entry for each domain variable. This entry can either be the domain itself or it can be a pointer to the domain.
- **Local link:** In this case we should use a data structure which physically combines the variable with its domain.

In FIDO-II a local approach has been preferred, since it avoids the search in the global table, which would have been a very frequent operation otherwise. A FIDO-II domain variable is a compound data structure, which is internally represented as follows:

Definition 11 (FIDO-II Domain Variables) *A domain variable X is a sixtuple*

$$X = (\&, Id_X, Length_X, Constraints_X, Val_X, Dom_X),$$

where Id_X, $Length_X$, $Constraints_X$, Val_X, Dom_X are variables.

The components of a domain variable X are described in the following:

- Its first argument, the ampersand, is a flag used to identify the sixtuple as a domain variable.
- Id_X is an atom denoting the unique identifier of the domain of the variable[6].
- Val_X actually incorporates the value of variable X. As long as Val_X is unbound, X is considered to be unbound. The value Val_X is finally bound to denotes the actual value of X.
- Dom_X carries the domain of X. A detailed discussion of domains in general and of the concrete structure of Dom_X in special can be found in the following chapter 5.1.3. Note, however, that it is a very cheap operation to access a variables domain if the variable is given, namely a simple variable reference. This is an important advantage of the local representation of domains in FIDO-II.
- Length_X is an open list whose last nonground element is interpreted as the actual number of valid domain elements, thus it represents the actual domain length. This information is needed for an efficient implementation of the *singleton test* (see chapter 6.2.3) on one hand, and it can be used for a first fail heuristics as regards variable instantiation order (see chapter 6.5).

[6] which must have been introduced before together with the variable itself by using the *define_domain*/3 predicate.

- Finally, Constraints_X shall contain the actual number of constraints a variable appears in. It shall also be implemented as an open list as it is described for Length_X. The number of constraints can be used for another variant of a first fail heuristics.

As regards the Constraints_X variable, I would like to mention that this is not supported by the current prototype implementation. In the initialization phase, Constraints_X is simply set to the open list [0|_] and ignored for further computation. However, I kept it in its place since this facilitates a later integration of a mechanism keeping track of the number of constraints for each domain variable, without having to change the internal integration representation of domain variables. Since many internal procedures refer to the actual structure of domain variables, such a change would cause great cost and a great deal of debugging, because all these procedures would have to be adapted to the new domain variable representation.
During the rest of the work, when I refer to domain variables and so to their components, I will use the names defined here, i.e. Val_X for the value of a variable X, Dom_X for its domain and so on.

5.2.2 The User's View on Domains

From the Users' point of view, in FIDO-II, domains have to be declared locally by a

```
define_domain(DomainID, VarSpec, DomSpec)
```

declaration. `DomainID` is a PROLOG atom identifying the domain. It can be useful in order to increase efficiency of some operations on domain variables. Each domain variable carries the identifier of its domain. Thus, e.g. the unification of two variables sharing the same domain can be implemented in a simple but efficient manner, since it can be reduced to standard PROLOG unification once it is known that the variables have the same domain. This, in change, can be detected very easily using the identifiers[7].
The second argument, `VarSpec` gives a definition of the domain variables used in the clause containing the `define_domain` call. `VarSpec` can be either

- A list of variables, which will be treated as domain variables by the preprocessor, or
- A predicate call of the form **gen_var(N, L)**, which is a FIDO library predicate creating a list L of N new domain variables, where N has to be an integer. In particular, N can receive its value at run-time, thus facilitating dynamic domain declarations.
  This way, it is possible to formulate e.g. a program for N queens with arbitrary N, as it is shown in figure 5.1.

The third argument of `define_domain / 3`, which is `DomSpec`, specifies the way the domain actually looks like. It can be of one of the following three forms:

1. A list of PROLOG ground terms, e.g. integers, atoms, strings. This offers a convenient way to express smaller domains simply by enumerating their elements.
2. An expression of the form **n..m**, where **.. / 2** is a FIDO-II operator which denotes the closed integer interval $[N, M]$ ranging from the integer N to the integer M. N must be smaller or equal than M, else an error is raised.

[7] A more serious problem arises if domains are not syntactically, but semantically equal. This will be discussed later on.

3. For reasons of convenience, user defined predicates are allowed in order to describe domains. If e.g. a programmer wants to create the domain of all prime numbers less than 1000, an explicit enumeration of this domain would be quite an awkward work. Instead, the user can write a PROLOG procedure computing the list **Primes** of prime numbers, for example with the call **erastothenes(Primes, 1, 1000)**, add the definition of **erastothenes / 3** to the program and write the procedure call as third argument into his domain definition. This is what happens in example 5.1, where the user-defined predicate **gen_int_dom / 3** creates an integer domain.

Here, some cautionary remarks should be made on the points 2 and 3:
Using the format $N..M$ for specifying domains, one should pay attention to the following:

- The operator ../2 may only be used in **define_domain / 3** declarations.
- N and M are not allowed to be variables, but must be integers.

Concerning domain generation by user-defined predicates, it has to be paid attention to two conventions:

- The first argument of the user-defined procedure must contain the list of domain elements resulting from the call. An arbitrary number of arguments may follow.
- The procedure creating the domain must be known to the PROLOG system at run time

```
n_queens(N, L) :-
    define_domain(queens, gen_var(N, L), gen_int_dom(D, 1, N)),
    safe(L),
    instantiate(L).   \* instantiate variables *\
```

Figure 5.1: Example: Top Level Definition of General N Queens Problem

5.2.3 An Example for the Use of `define_domain / 3`

In this short paragraph, certain ways of representing *one* problem are shown. Example 5.3 describes three equivalent manners of writing down the definition of a domain for the *four queens* domain, which contains the elements $\{1,2,3,4\}$.
The first is by explicitly enumerating the domain elements, the second by defining the respective integer subset and the third is by using a self-defined predicate **gen_int_dom / 3** in order to create the list of domain elements. **gen_int_dom / 3** itself is implemented as shown in figure 5.2.3.

Although being semantically equivalent, the use of different domain declaration formulations provides to the system different additional information. This can lead to differences concerning efficiency of operations executed on the domains, e.g. the way a forward-checking algorithm accesses domain elements. A more detailed discussion of this fact can be found in the following chapter 5.2.4.

```
\* gen_int_dom(D, N, M) succeeds if D is the lists of integers from N to M *\

    gen_int_dom([], N, M) :-
        N > M,                  \* stop condition *\
        !.

    gen_int_dom([N|T], N, M) :-
        N =< M,
        N1 is N + 1,
        gen_int_dom(T, N1, M).
```

Figure 5.2: Example of a User Defined Predicate Creating a Domain

```
queens([X1, X2, X3, X4]) :-
    define_domain(queens, [X1,X2,X3,X4], [1, 2, 3, 4]),
    ....

queens([X1, X2, X3, X4]) :-
    define_domain(queens, [X1, X2, X3, X4], 1..4),
    ....

queens([X1, X2, X3, X4]) :-
    define_domain(queens, [X1, X2, X3, X4], gen_int_dom(D, 1, 4)),
    ....
```

Figure 5.3: Example: Different Possibilities to Formulate a Domain Definition

5.2.4 Using Domain Specific Information

As we will see in chapter 6.2.2, the efficiency of consistency algorithms for constraint propagation on finite domains crucially depends on the knowledge available about the structure of the specific domains. Thus, e.g. information about orderings on domains can be exploited for some constraints, e.g. the $<$ and the $>$ constraint. Besides ordering information (is the domain ascendingly or descendingly ordered?), knowledge about the actual domain length or about the possibility to compute the adress of single domain elements to provide direct access to them (which depends from the way the domain is stored and from the domain itself)[8] can be used.
This information can be gained quite simply if the contents of the domain are known at compile-time. It is, however, more difficult once we allow dynamic domain definitions, as can be achieved by user-defined predicates creating the domains in FIDO-II (see chapter 5.2.2). In this case, at compile-time, the system does not know anything about the domains. Thus, an examination of the domain structure has to be performed at run-time, slowing down the system. Otherwise, advanced domain-specific access methods cannott be used, slowing down the system, too.
In any case, it would be of great advantage if additional information (e.g. concerning domain ordering) could be formulated and added to the program by the user.
In the current prototype implementation of FIDO-II, a restricted possibility to do so is pro-

[8] e.g. for integer subset domains stored as ordered lists, the offset of each domain value is computable. This will be used in the prototype implementation

vided. The programmer can formulate a fact **domain_order(Domain_Id, Order, Offset, Length)**, which is handled as a declaration. The arguments are explained in the following:

- **Domain_Id** denotes the identifier of the domain to which the declaration refers.
- **Order** can be either *asc* for ascendingly ordered domain or *desc* for descendingly ordered domain w.r.t. an arbitrary order intended by the user.
- **Offset** can be either variable, or it can be an integer, which denotes the integer offset of each domain value in the domain **Domain_Id**. If it is variable, it is ignored by FIDO.
- **Length:** This variable can be given the length of the domain. This is only required if the domain is created by a user-defined predicate. Otherwise, or if the value is variable, it is ignored by the system.

If any arguments of `domain_order` /4 are left uninstantiated, FIDO-II treats them as unknown. If no declaration at all is added for a domain, the system assumes worst-case, which means sequential search within the domain. However, there is an important exception from this: Domains declared as an integer interval by a domain specification of the form $N..M$ are represented and treated as ascendingly ordered, and the offset O of a domain element D_i can be computed as $O = D_i - N + 1$. This facilitates a direct access to domain elements. In order to give a feel for the consequences of that, table 5.4 compares the run-time for the N queens program using a domain description of the form $1..M$ with the run-time for the program obtained by creating the domain with the help of the user-defined predicate `gen_int_dom / 3` as defined in chapter 5.2.3.

N	Specialized Decl.(S)	General Decl.(G)	S/ G
4	0.07	0.07	1.0
8	0.53	1.12	0.47
12	2.23	4.18	0.53
16	0.83	2.08	0.39
32	3.52	12.35	0.29
48	10.2	59.0	0.17

Figure 5.4: Runtime of N Queens Depending on Domain Declaration

Obviously, using domain specific information for the algorithms accessing the domains yields a remarkable improvement in run-time performance. However, it should be mentioned that the considerable difference in performance appearing here is nothing more than a shortcoming of the actual prototype implementation, since it goes against the paradigm of *data independence*: Actually, (e.g. as it is the case in declarative data base queries (for example, SQL)) the performance (answering time) of the system should be largely independent from the way the problem (here: the domains, in database systems: the query) is formulated[9]. A thorough discussion of the notion of data independence can be found in [Hä87].
A more sophisticated implementation of FIDO should nivellate these differences in performance caused by using different, but semantically equivalent domain definitions. This could be achieved by an extended program analysis during precompilation, or, if this is not possible, at run-time (which, however, causes some overhead).

[9]This could also be looked upon as a further instance of the declarativity / procedurality competition.

5.2.5 Internal Realization of Domains in FIDO-II

After weighing the points mentioned in chapter 5.1.3 and chapter 5.1.4, and after experimenting a bit with some representationsof domains, I decided to use a mixture between tree and list representation for domains, classifying valid from invalid domain elements by a flag.
A domain **dom** with N elements is represented as a flat uninterpreted functor, say f_{dom} with N arguments $Arg_1,\ldots, Arg_N$. Each argument Arg_i , $1 \leq i \leq N$, consists of a 2-tuple (Val_i, $Flag_i$), where Val_i is a ground PROLOG term denoting the actual domain element. $Flag_i$ is a variable which is instantiated to 0 if the element Val_i is eliminated from the domain. Since to each variable its domain is physically attached, the possibility of undoing the flag instantiations on backtracking is guaranteed. Thus, domain restrictions based on assumptions that have turned out to be inconsistent with the current state of the constraint net can be undone.

Since domains have the form of uninterpreted functors, domain elements can be directly accessed using the PROLOG **arg / 3** built-in predicate, provided that their position within the domain (their adress) can be easily computed (see chapter 5.2.4). Figure 5.5 shows the internal representation of sequential lists and general N-ary functors on the PROLOG heap, giving a feel for the qualitative differences as regards access to single elements of these data-structures. Moreover, by iterating **arg / 3** over the domain length, a sequential access is possible, which is certainly slower than sequential access via lists, since the current argument counter has to be incremented once for each domain value, which involves an arithmetic operation. However, due to the flat structure of domains, deeper recursion[10] can be avoided. Thus the overhead will not exceed a constant factor here, making the trade-off between having fast sequential and direct access to domains still reasonable. Certainly, we cannot claim that the representation chosen in FIDO-II is *always* superior to a list representation, but, in many cases, the advantage of cheap direct access seems to be worth the aditional cost for sequential domain traversal.

5.2.6 Concluding Remarks

In this subchapter, we have seen how domains are brought about in FIDO. I will finish with some concluding words. First, I would like to add just a few remarks on the use of the **define_domain / 3** predicate for defining domains:
This predicate is used in the clauses where domain variables are desired. Certainly, as the application examples in chapter 7 show, each domain has to be declared only once for the use within a program, i.e. in the top level clause of this program. However you should be careful of using programs providing recursive calls to procedures containing **define_domain** goals, since every call to this predicate will create the domain from scratch, thus, destroying currently existing versions of it.
Note also that the domain definitions are local ones, i.e. they are valid only within the clause containing them. Defining a domain with the same domain identifier twice within different clauses may lead to problems, since only the most recent domain definition will survive. Therefore, for different domains, different names should be used.
As I said above, FIDO-II uses local domain definitions, whereas CHIP e.g. describes domains by global declarations. The local solution has the advantage of allowing an explicit identification of the domain variables appearing within a program. This is avoided in CHIP, since global variables are not really compatible with pure PROLOG semantics. It is my opinion

[10] that would be necessary if general tree structures were used.

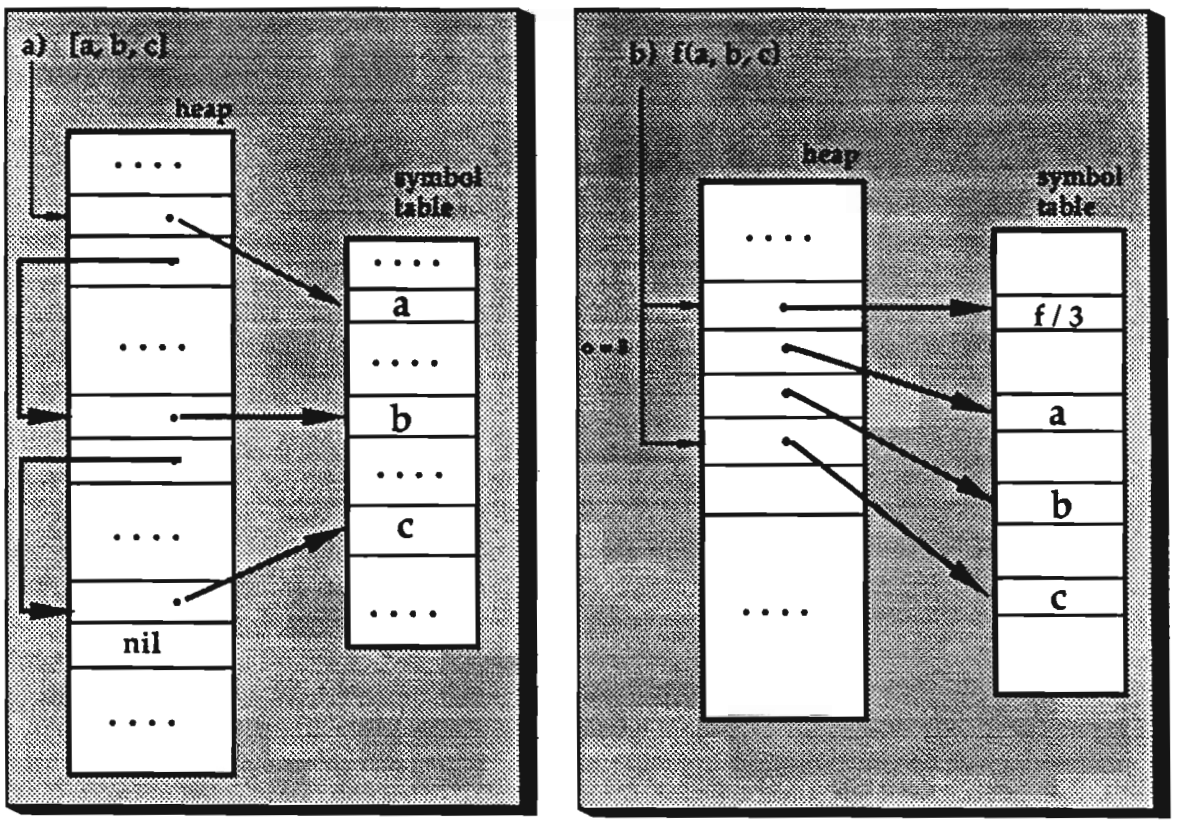

Figure 5.5: Example: Representation of Lists (a) and Functors (b) on the PROLOG Heap

that the local definitions contribute to an easier understanding of the operational program behaviour (you see where the domain variables appear and what happens to them), preserving the standard PROLOG semantics. Especially, in clauses that don't use the domain extensions no overhead for a global lookup is required.

5.3 SEPIA Meta-terms: a Simpler Way of Maintaining Domain Variables

Since version 3.0.16 [SEP91], SEPIA PROLOG offers a feature by which a more convenient (and maybe also more efficient) implementation of domains and their maintenance can be achieved. This is the data structure of **meta-terms**. In this paragraph, I will briefly describe the general idea of this feature, and some considerations towards an implementation of domains using meta-terms will be made.

5.3.1 Meta-terms

Unification is the kernel of each PROLOG system. Up to now, in SEPIA, it had been the only part of the system that could not be modified.
This is remedied in the current SEPIA version 3.0.16 by the implementation of a new data type, **meta-terms**. By using meta-terms, the programmer can define new generic types and data-structures, and, what is even more important in our case, he can specify how they should be treated by the system (i.e. by built-in predicates). That way, unification of domain variables can be redefined.
From a logical point of view, a meta-term is considered as a variable. The difference to "normal" variables is that an attribute can be associated with the variable. This attribute can be any PROLOG term. Whenever the PROLOG machine encouters a meta-term, an event is raised which depends on the type of operation to be performed on the meta-term. These events can be redefined by the user. Thus, we have a facility of controlling the operations on meta-terms performed by the system. That particularly provides us with a *hook* for unification, i.e. an opportunity to recognize and modify it. For a more detailed description of meta-terms, I refer to the SEPIA 3.0.16 user manual [SEP91].

5.3.2 Using Meta-terms in FIDO-II

The possibility of attaching an attribute to a variable can be very useful for the implementation of domain variables ranging over finite domains. As I showed in chapter 5.2.1, a FIDO domain variable is represented as an explicit sixtuple containing the information needed to maintain the domain variables. According to section 6.2.5.1, the fact that these variables are not considered as variables by the underlying PROLOG system, leads to problems as regards domain variable unification. This is particularly true for implicit unification caused by unifying clause heads.
Since meta-terms *are* recognized and handled as variables by the system, they offer a way to maintain domain variables in a deper system layer, leading to more consistent and efficient performance. Example 5.6 shows how domain variable unification can be implemented using SEPIA meta-terms for the case of two domain variables to be unified. Event 10 is raised if two meta-terms are unified, i.e. if the predicate = /2 is applied (implicitly or explicitly) to meta-terms. The handler of this event is set to the predicate `dv_unify` /2.
Note that the singleton test becomes very simple in the above example. Backtrackability is guaranteed without having to use flags in order to remove domain elements. Thus, all the problems I had to face due to the explicit domain variable representation in FIDO can be avoided. This allows a more elegant implementation that, however, requires much less programming effort. Since domain variable unification can be left to built-in predicates, I suppose that a system based upon meta-terms could even run a little faster than it actually does.

```
?- set_error_handler(10, dv_unify / 2).

dv_unify(X, Y) :-
    meta_term(X, DomX),                 \* both X and Y are *\
    meta_term(Y, DomY),                 \* domain variables *\
    intersection(DomX, DomY, DomZ),     \* compute domain intersection *\
    (
      if                                \* intersection singleton? *\
        DomZ = [Val]
      then                              \* yes! *\
        meta_bind(X, Val),              \* bind X and Y to the \*
        meta_bind(Y, Val)               \*     singleton value *\
      else                              \* no! *\
        meta_term(Z, DomZ),             \* create new variable ranging over *\
        meta_bind(X, Z),                \* the intersection of X and Y and *\
        meta_bind(Y, Z)                 \* bind X, Y to that new variable *\
    ).
```

Figure 5.6: Implementing Domain Variable Unification Using SEPIA Meta-terms

The problems I see are mainly the lacking experience concerning the stability of the system and the generalization of the algorithms for arbitrary constraints. It would be necessary to have a closer look at these issues when starting to implement domain variables based upon meta-terms.

Chapter 6

The Integration of Control

In this chapter, I would like to describe how an advanced (forward-checking or looking-ahead) control can be achieved in FIDO-II. After having a closer look at "constraints" in FIDO-II, the implementation of consistency techniques will be discussed.
After that, the static structure and the dynamic behaviour of the FIDO-II preprocessor will be outlined. In a further subchapter, I will show some problems revealing limitations of the approach, joined with a short outlook.

6.1 Motivation

A major goal of logic programming research is to build systems that will *efficiently* solve problems stated in a convenient, *declarative* way. The words *efficiently* and *declarative* turned out to be quite antagonistic, symbolizing the well-known gap between the paradigms of procedurality and declarativity. That has been initiated by the famous paper of Winograd [Win75]. Both these research paradigms have gathered large communities around them. A nice recent summary of this topic can be found in [RB91].
But what is the actual reason for this struggle? Formulating tasks in a declarative way is certainly a good thing, because it is much more convenient[1] to formulate a problem by simply writing *what* shall actually be done than by describing *how* it is to be performed.
But, unfortunately, executing a problem is a different kettle of fish, because the machine certainly has to know *how* a problem shall be solved. As experience with logic programming languages shows, direct transformation of declaratively formulated programs into machine code (e.g. using SLD resolution as done by PROLOG) turns out to be of a latent inefficiency. In my opinion, approaches which provide special machine architectures for special declarative languages as PROLOG, trying to force the machine to 'think' in declarative structures, start at the wrong end. So why shouldn't we write programs in a declarative style, but execute them following the procedural paradigm, charging the machine with the task of finding the best (or at least a good) algorithm for solving the problem? A fact that fortifies this belief is that one of the fastest recent PROLOG systems designed and implemented by Taylor [Tay91] essentially relies on a preprocessing phase based on global analysis and program transformation. This means finding out *what* the user actually wants and reformulating it in a way which is appropriate to the actual machine (which is a general purpose RISC machine in Taylor's approach).
For me, the key to close the gap between declarative formulation and procedural execution seems to be a compilation mechanism, which solves the '*how?*' question in an intelligent (why

[1] and reduces the probability of making mistakes, as well

not procedural?) way starting with information about the '*what?*'.[2]
A crucial point in this context is the choice of appropriate, intelligent control strategies. In this chapter, we will see how that can be achieved for PROLOG using a coroutining mechanism. I will refer to the motivation and the definition of the relevant notions such as constraints, forward-checking, and looking-ahead given in the previous chapters 1, 3.

6.2 Constraints in FIDO-II

As we saw in chapter 3, the definition of a constraint is a very general one. Doubtlessly, this generality should be reflected by an implementation. Seen from a more practical point of view, however, some constraints happen to be more important than others. These frequently used constraints are e.g. constraints expressing various equality relations (subsuming PROLOG unification - what a mighty concept!), inequality, arithmetic constraints and constraints referring to set orders, such as >, <, =< or >=. The efficient implementation of these constraints, strictly speaking of the effects of their execution on the domains of the variables involved is very important for a satisfactory performance of the system.

6.2.1 Classifying Constraints

There are various ways of classifying constraints. In [van89a], a classification in terms of *elementary* and *composite* constraints is proposed. Van Hentenryck also introduces a distinction of constraints depending on how they are used, i.e. whether as *tests* or as *choices*. In this work I made a different distinction which is motivated by the pragmatic separation between *more important* and *less important* constraints mentioned above. I introduced two general types of constraints as they appear from the perspective of the system. The first type is called **built-in** constraints, the second type **user-defined** constraints. Both types will be described in the following.

6.2.1.1 Built-in Constraints

Basically, there exist a great number of constraints worth being implemented in a particularly efficient manner, i.e. as built-in constraints. To these belong various arithmetic constraints, e.g. `plus / 3`, `times / 3`, but also comparison constraints, which include various forms of equality and the ordering relations. Besides, other constraints as e.g. `min / 3`, `max / 3` which find the minimum resp. maximum of two elements or the `element / 3` constraint which, spoken procedurally, computes the Nth element of a list, can be considered worth a support by the system, since they are used quite frequently. In a few words, the list could be extended almost infinitely.
In the current prototype implementation of FIDO-II, I restricted the number of built-in constraints to six. Built-in constraints are

- equality constraints:
  - the unifiability constraint = /2
  - the numeric equality constraint =:= /2

[2] Amen!

- inequality constraints:
  - the non-unifiability constraint \= /2
  - the numeric inequality constraint =\= /2
- ordering constraints:
  - the greater-than constraint > /2
  - the less-than constraint < /2

I made that choice, because many other constraints can be reformulated in terms of these constraints[3] For the built-in constraints, specialized versions of forward-checking (and for some of them for weak looking-ahead, too) are implemented, trying to increase performance for programs using the built-ins.
A more detailed discussion of this topic can be read later in this chapter, when the realization of consistency techniques is described.

6.2.1.2 User-defined Constraints

In order to reflect the generality of the constraint notion, a second class of constraints is introduced: the so called user-defined constraints.
Every predicate which is provided with a **forward** declaration (see section 6.2.3) and which is not a built-in constraint, will be handled as such a general, user-defined constraint. That is, a general consistency algorithm will be applied to it. Since the general algorithm (e.g. for forward-checking) does not utilize information about special properties of the constraints w.r.t. the underlying domains, user-defined constraints are normally less efficient than their built-in counterparts.

6.2.1.3 The User's View on Constraints

From the semantical point of view, the distinction between built-in and user-defined constraints can be regarded irrelevant for the user. A predicate **ne / 2** which is defined as

```
ne(X, Y) :-
    X \= Y.
```

has the same semantical behaviour as the direct call to the forward-checking version of the built-in constraint \= /2, if it is used together with a **forward** declaration. But, suppose X, Y are domain variables, there will be a difference in efficiency, since the use of the built-in inequality constraint will exploit constraint specific information[4] and thus, it will yield better performance than the call to to the forward user-defined constraint **ne** /2.
Thus, in terms of efficiency, the user actually should be aware of the two constraint types. I consider this as a shortcoming of the system, but it is one caused by the current implementation, not a general one. By providing a greater number of built-in constraints and by a more

[3] and certainly because I didn't have three years of time

[4] This information consists of the fact that at most *one* domain value will be eliminated by *one* forward-checking test using \= /2.

sophisticated static program analysis, it would be possible to yield an automatic recognition and maintenance of constraint use, having the system find the optimal (or at least a good) algorithm depending on the actual constraints and domains. For example, the system could try to replace some calls to user-defined constraints by calls to built-in constraints, as it would be possible with a constraint $X =< Y$ (X, Y be integers), which could be replaced by the constraint $X < Y + 1$, taking care of the case when Val_Y is the maximal element of Dom_Y, i.e. when $Y + 1$ is not defined.

6.2.2 Consistency Techniques: A Motivation

In this chapter the realization of advanced control strategies in FIDO-II is described. Here, emphasis is placed on forward-checking.
Constraint solving is a well-understood technique in the area of AI. Various tools for propagating information (i.e. variable values or sets of values) through constraint networks and various programming languages supporting constraints exist (see e.g. Sketchpad [Sut63](!!), ThingLab [Bor79, Bor85a, Bor85b], TK!Solver [KJ84] or JUNO [Nel84] as examples of systems using numeric techniques, [Ste80] or IDEAL [vW80] as examples of systems using symbolic techniques). Two major methods for ensuring local consistency are *forward-checking* [HE80] and *looking-ahead*. The necessary theoretic principles providing local arc-consistency have been pointed out in chapter 3. As we saw in chapter 1, due to their a priori use allowing active reduction of search spaces, these techniques seem to form an ideal completion of the poor *generate & test* or *standard backtracking* search strategies supplied by standard logic programming, say PROLOG. It is generally believed [ED89] that, for most applications, forward-checking is the adequate means, for it combines efficient search space pruning with an overhead which is mostly acceptable. On the contrary, looking-ahead is often regarded as being too inefficient (since causing too much overhead).
Therefore, in the prototype implementation of FIDO-II, the stress lies upon forward-checking. looking-ahead is only implemented in a slightly modified version for some distinguished constraints.

6.2.3 Forward-Checking in FIDO-II

The formal definition of forward-checking, which is caught by the forward-checking inference rule (FCIR) [van89b], has been given in chapter 3. Informally, constraints can be used in a forward-checking manner as soon as only one of their domain variables is uninstantiated. Then, the set of possible values that can be given to this last variable is reduced.

6.2.3.1 Forward Declarations in FIDO-II

In this paragraph the user's notion of forward-checking shall be described. The programmer can enforce a forward-checking use of a predicate p/N within a goal by a local declaration using the predicate

forward /1.

`forward` /1 has as its only argument the constraint that shall be executed by forward-checking. Example 6.1 shows how forward-checking can be used to implement the `regular` /3 predicate

```
regular(X, M, Y) :-
    forward(Y =\= X),          \*  (1)  *\
    forward(Y =\= X + M),      \*  (2)  *\
    forward(Y =\= X - M).      \*  (3)  *\
```

Figure 6.1: Example: Constraint Definition for the N Queens Problem

which is used to state the constraints in the N queens program (See figure 7.1 for the complete program source).

`regular(`X`,` M`,` Y`)` gives expression to the condition that queen X, which is to be placed in column i, $1 \leq i \leq N$, and queen Y which is to be placed in column $i + M$, must not threaten each other. i.e. the queens X and Y are neither allowed to stand in the same row, which is denoted by constraint (1) in the example, nor they may be put on a diagonal (as expressed by the constraints (2) and (3)). $=\backslash=$ is the numerical inequality predicate, which evaluates its arguments. The goal $5 \ =\backslash= 3 + 2$ fails, whereas e.g. $5 \ \backslash=3 + 2$, which only checks non-unifiability, succeeds.
What is new compared with the way a generate & test or standard backtracking program works, is how the FCIR prunes the search space:
Assume that, in the above example, queen X in column c_1 has become instantiated with a value i, i.e. row i has been assigned to that queen. The call `regular(`X`,` M`,` Y`)` (where N is an integer) will have an a priori pruning effect on the remaining possibilities of placing queen Y in column $c_2 := c_1 + M$. This is shown in figure 6.2 with $c_1 = 2$, $i = 3$ and $M = 2$.

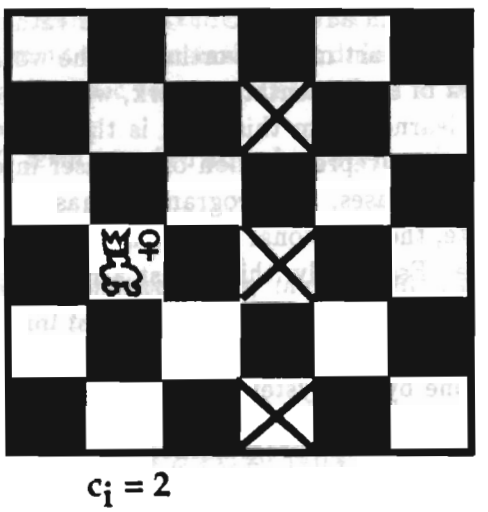

Figure 6.2: The A Priori Pruning Effect of Forward-Checking for 6 Queens

Here, three values can be eliminated from the domain of Y.

6.2.3.2 Arguments of Forward Declared Constraints

Figure 6.1 shows two differences between the representation chosen here and the way *forward* declarations are made in CHIP[DvHS+88]. The first thing is that CHIP uses global forward

declarations. The second difference concerns the amount of information the user can, respectively should include into the declaration. In FIDO-II, the programmer does not have to make assertions about the arguments w.r.t. being domain variables[5] or ground. In CHIP [van89a], the arguments of constraints must be labeled as domain variables or as ground by a flag inside the `domain` declaration, which can be 'd' respectively 'g'. The semantics of a CHIP declaration `regular(d, g, d)` is the following:
The FCIR can be applied to any call to the predicate `regular / 3` if

- its second argument is ground, and
- `regular / 3` is forward-checkable w.r.t. the elements corresponding to a 'd' in the declaration, i.e. the first and the third argument.

Enhancing the declaration by additional information in CHIP makes life easier for the system, since only the arguments labeled by a 'd' can be domain variables and must be treated that way. In FIDO-II, however, it must be checked for all arguments whether they can contain domain variables. This requires a thorough static program analysis at precompile time and decreases run-time performance, since several case-distinctions must be made. Besides, in the current implementation, the FIDO-II preprocessor can figure out at compile time which constraint arguments *might* be instantiated with domain variables, but not, which ones actually *will* be at run time. Thus, the information resulting from static analysis does not lead to wrong computation results, it simply tends to be too cautious in some cases, making the domain variable test too often, as I will show in section 6.6.
The disadvantage of the CHIP representation is that it enforces the programmer to go deeper into describing the procedural behaviour of the program. He has to tell the system which arguments can (or will) be called with domain variable arguments. Certainly, it is a matter of opinion whether this is an advantage of CHIP or rather a shortcoming, whether the gain of convenience on the user's part makes worthwhile the work to be done (even at run-time), and the additional amount of implementation work, which is considerable.
One of the lessons I learned from this work is the following: if I had to start from scratch, I would probably choose a representation of the user interface to FIDO-II similar to the one in CHIP. Since, in most cases, the programmer has a more or less clear idea of the calling patterns of a predicate, the additional trouble he has to undertake in order to declare it to the system seems tenable. Essentially, this is just a remake of the well-known trade-off between two antagonistic questions:

- What *can* be done by the system? and
- What *must* be done by the programmer?

However, allowing the user to integrate knowledge about the calling patterns, but not forcing him to do so could be a good compromise[6]. If he does not add information, then the system itself should master the task of finding possible calling patterns. This would answer the two questions above in a very simple way:
The system has to do the things the programmer himself cannot (or doesn't want to) do!

[5] by the way, even if he would like to, he couldn't do so. This is certainly another shortcoming of the system. See chapter 8.2 for this.

[6] I really don't know whether this is possible in CHIP

6.2.3.3 Formats of Constraint Arguments in FIDO-II

Let me come back now to the format of constraint arguments themselves. In FIDO-II, any argument of a constraint must have one of the following formats, depending on the constraint type:

- If the constraint is a built-in, it must be of a *simple polynomic* form
- If it is a user-defined constraint, it must be *simple*

In the following, the formats used in italics above will be defined.

Definition 12 *A PROLOG term is called* **simple**, *if it has exactly one of the three forms*

- *A PROLOG variable*
- *A domain variable*
- *An atomic expression, i.e. an atom, a string or an integer.*

Definition 13 *A PROLOG term is called to be of a* **simple conjunctive** *form , if it has exactly one of the two forms*

- *a simple term*
- *Be X, Y simple conjunctive terms, then $(X * Y)$ is again a simple conjunctive term*

Note that Definition 13 only defines syntactically correct terms. Whether such a term makes sense if it is used as an argument of a constraint is a different question [7].
Now, we can define simple polynomic terms as disjunctions of simple conjunctive terms:

Definition 14 *A PROLOG term is called* **simple polynomic**, *if it is*

- *simple conjunctive, or*
- *Be B_1, ..., B_n simple conjunctive terms, then $\oplus_1 B_1 \oplus_2 \ldots \oplus_n B_n$, $\oplus_i \in \{+, -\}$, is again a simple polynomial term*

An example for a simple polynomic term t is:

$$t \equiv -(A * B * C) + (10 * Z) - 5 \; .$$

Parentheses are normally left out for convenience.

Thus, a convenient way to state a large class of arithmetic constraints is provided. Internally, during preprocessing, these expressions are transformed by a normalizer into a form that can be handled by the forward-checking mechanism. For this I refer to the next section 6.3.
The recognition of the predicates that actually use domain variables is left to the system, which performs a static data flow analysis in order to find out which predicates are possibly called with domain variable arguments.

[7] e.g. the term "zwei" * 3 is a syntactically correct simple conjunctive term, even it is not appropriate due to the interpretation of the */3 predicate.

6.2.3.4 The General Forward-Checking Algorithm in FIDO-II

In this paragraph the realization of a forward-checking algorithm for arbitrary constraints is described. Figure 6.3 shows how the algorithm is implemented in SEPIA PROLOG. Let us

```
\*   Semantics of the arguments                                          *\
\*   Dom    = Domain of the forward Variable VAR                         *\
\*   Call   = Original Constraint Call                                   *\
\*   N      = Arity of the DOM term (maximal domain length)              *\
\*   Length = Actual domain length (number of valid elements)           *\
\*   Argnum = Number of the argument of DOM containing the forward variable *\


gen_fc(Dom, Var, Call, Length, Argnum) :-
    functor(Dom, _, N),
    gen_fc_restrict(Dom, N, Call, Argnum, Length, 1),
    !,
    gen_singleton_test(Dom, Length, Var).



gen_fc_restrict(Dom, N, Call, Argnum, Length, M) :-
    M > N,
    !.

gen_fc_restrict(Dom, N, Call, Argnum, Length, M) :-
    arg(M, Dom, (Val, Flag)),       \* Access to Mth domain element *\
    (
     if
       Flag \== 0
     then                           \* Domain element is valid *\
       copy_term(Call, Call1),      \* create copy of term with new var's *\
       arg(Argnum, Call1, Val),     \* insert value VAL into the *\
       (                            \* corresponding place *\
        if
          Call1
        then                        \* Constraint satisfied *\
          true
        else                        \* Constraint not satisfied: *\
          Flag = 0,                 \* --> remove domain element *\
          dec_length(Length)        \* increment domain length by one *\
     else
      true
    ),
    M1 is M + 1,                    \* test rest of domain *\
    gen_fc_restrict(Dom, N, Call, Argnum, Length, M1).
```

Figure 6.3: Example: A General Forward-Checking Algorithm

have a look now at what the algorithm actually does:
it is activated by a call to the `gen_fc / 5` predicate. A description of the arguments can be found in the documentation of example 6.3. The algorithm runs in two phases. The first phase consists of a sequential scan of the domain `Dom`, trying for each valid domain element `Val` whether it satisfies the constraint `Call` if used as value for the forward variable. If an element does not, it will be eliminated from the domain by instantiating its validity flag to 0. In this case, the domain length variable is updated by decrementing its value.

The second phase is the so called singleton test. After every forward-checking, it has to be tested which of the following three cases is true. The domain can contain

- more than one valid element: nothing happens, computation goes on.
- exactly one element: this element must be found and assigned to the domain variable `Var`.
- no more valid elements: The `gen_fc / 5` call (and so the constraint itself) fails.

In order to decrease cost for the singleton test, it is important to offer a cheap way to access the actual domain length, which is stored in Length_X for each variable X.
As I showed in section 5.2.1, Length_X is presented as an open list, whose last valid element is an integer representing the actual number of admissible domain values. Thus, at the beginning of each forward-checking call, Length_X has to be traversed once in order to get the acual length of the domain.

Efficiency Considerations From a logical point of view, *one* call of `gen_fc / 5` can be looked upon as performing an atomic operation. Since this implies that on backtracking, all changes made in it will be revised, it is not necessary to keep track of each deletion of a domain value within this call by decreasing and restoring domain length [8]. Rather,the initial value can be locally decreased, then be used for the singleton test and finally be rewritten as new value of Length_X.
This way, the cost for maintaining domain-length can be limited to a tenable degree, even if programming it in PROLOG. If we assume that the average length of the length list of a domain with cardinality N is N / 2 (this corresponds to the situation after half the domain values are eliminated), the cost computes to $2*(N/2)$ list element accesses per forward-checking call. In the worst case $2*N$ accesses are necessary. The cost for the singleton test can be kept low by using the actual domain length. If the domain is not singleton, this can be detected by only one comparison. If it is singleton, the last remaining domain value will be found by average $N/2$ list comparison operations (N in the worst case).
It is clear that the price to pay for the generality of the general forward-checking algorithm is a minor efficiency. This is caused by the fact that no specific knowledge is used about peculiarities of the predicates w.r.t. the domains. As we will see in the next subsection 6.2.3.5, the availability of such kind of knowledge facilitates much more efficient algorithms.

6.2.3.5 Using Specialized Forward-Checking Algorithms for Built-in Constraints

The notion of built-in constraints makes it possible for the system to master the forward-checking task in a more efficient way. For each built-in constraint and each relevant domain type, a specialized version of forward-checking is implemented, which uses knowledge about specific behaviours of these constraint w.r.t. different domains.

Domain Types In FIDO-II, several criteria can be used to classify domain types. This classification is performed at precompile time whenever possible[9]. During preprocessing, it is kept track of which predicates are called with domain variables as arguments, also storing the

[8] which would mean one traversal of the length list for every value to be eliminated

[9] This is not possible in all cases, since FIDO-II gives support to dynamic domain definitions (see section 5.2.2).

domain identifiers of the domains used.
If any information is available about the domains, it can be used by the preprocessor in order to find and load an appropriate definition of the forward-checking algorithm to be used in the actual case. The following criteria are examined by the preprocessor:

1. Is there information available about a physical order underlying to the domain, i.e. ascending, descending or none ?
2. Is the arity of the functor representing the domain (i.e. the all over domain length) known at compile-time, or has it to be computed dynamically ?
3. Is an offset computable for the domain values, i.e. can the address of a domain element be computed by its value ?

The answers to these questions can be gathered by looking at the **define-domain** /3 goals defining the occuring domains, or by examining the domains themselves. According to this information, a suitable algorithm is chosen and code is added in order to load the definition of this algorithm.
The way the information is actually used depends on the constraints themselves. It will be outlined in the following paragraph.

Forward-Checking for FIDO-II Built-In Constraints In this paragraph, the principles of forward-checking use of the three classes of built-in constraints[10] is sketched w.r.t. the integration of knowledge concerning the behaviour of these constraints on certain domain types.

Equality If the user decides on using an equality constraint in a forward-checking manner, the following has to be done in the respective forward-checking call, which shall be `forward`$(X = Y)$, for example:

Assume Y is ground, and the domain Dom_X of X is to be restricted by eliminating all the values which are not equal to Val_Y. Practically, this task can be solved by looking up the value Val_Y of Y in Dom_X, instantiating X to that value, if Val_Y is a valid element of Dom_X, or stopping with a failure, otherwise. Since this processing is logically regarded as *one* atomic action (also concerning backtracking), domain length and domain flags actually don't have to be recomputed. As soon as a domain variable is instantiated, noone ever looks at domains, and if the binding must be revised after a failure, so are all changes concerning the domain made in that context.

Now let us see how efficiency of forward-checking equality constraints can be increased using pieces of information described above (see paragraph 6.2.3.5), coming back to our example:
If the offset of the value Val_Y within the domain of X can be computed, the test whether Dom_X contains that value happens to be very simple. If the offset is within the interval allowed for referencing the term Dom_X (which ranges from 1 to the arity of Dom_X), the value can be accessed by using the PROLOG **arg** /3 predicate with the address computed.
If this is not possible, sequential search within the domain must be started, until the value is found or it is sure that it is not contained in the domain. As regards the latter, order information can be used to detect failure earlier. If the domain is e.g. ordered ascendingly, search can be stopped after the first element has occured with a value greater than the value

[10] These are equality, inequality and ordering constraints.

to be retrieved. For descending order it works vice versa.
There are several forward-checking algorithms for equality provided. The decision about which ones to choose in a special case is taken by the preprocessor, depending on the domain-specific information available.
Let me say one more word concerning the forward-checking use of equality. To me, equality constraints seem to be the class of constraints to be handled most difficult. Especially, I think it is no good idea at all to use them in a forward-checking manner. In order to be forward-checkable, at least one of the two arguments must be ground. Equality, however, is such a strong constraint, providing strong relations between objects, that it should be used earlier in order to restrict search spaces. Therefore, I refer to section 6.2.5 where equality-specific problems are handled.

Inequality Forward-checking of inequality can be considered similar to the way equality is handled. The only difference is that, coming back to the notation used above, the inequality value Val_Y in a constraint $X \backslash= Y$, Y be ground, has to be removed from Dom_X instead of instantiating Dom_X to it, as done in the case of equality. Therefore, the considerations made for equality can be applied here, too.
Figure 6.4 shows the PROLOG-Code of two different forward-checking algorithms for inequality, handling two cases. Case 1, the procedure `fc_ne_o_1 / 5`, can be applied if the domain is ordered ascendingly, the offset of domain values is available, and the domain arity is given as an argument to the forward-checking call. As I pointed out in section 5.2.4, FIDO can access this information, if the domain has been specified by an expression of the form $N \ldots M$. For example, `fc_ne_o_1 / 5` can be used for the inequality constraints of the Five-Houses Problem, described in section 7.1.2. The FIDO-II source code of this problem is shown in figure 7.2. Figure 6.4 shows the forward-checking algorithm for an even more special case, i.e. if the address offset of all values is equal to zero. This corresponds to a domain, where the first element is 1, the second is 2 and so on. In this case, the value Val_Y can be directly used as first argument of the `arg / 3` access predicate, without any further computation necessary.

Generally, inequality constraints are predestinated for forward-checking use [van89b], since they allow straightforward elimination of domain values. Problems such as the N queens problem, where only inequality constraints appear, can be solved very efficiently by forward-checking, as it is shown in section 7.1.1.

Ordering Constraints Concerning efficient forward-checking use of the $>$ and $<$ constraint, it turns out to be crucial to have an order underlying to the domains, since this is the precondition for using these constraints in a way that physically refers to domain intervals, thus allowing efficient processing methods. There are two things varying, depending on the underlying domain order and the constraint itself:

- The starting point of search (i.e. from the last or from the first domain element).
- The exit condition (when can search be stopped?).

Assume for example, we want to use the constraint $X < Y$, with X ranging over $\{1, 2, \ldots, N\}$,and Y bound to the value $Val_Y = M \in \mathcal{N}$, in a forward-checking manner. We assume that the domain is physically ordered ascendingly. In the following, I will refer to the validity flag of the ith element of the domain Dom_X with $Dom_X[i]$.Flag and to the value of the ith element itself with $Dom_X[i]$.Value . Figure 6.5 shows the forward-checking algorithm used in this case.

```
\* Specialized forward-checking algorithm for \= / 2 and =\= / 2 constraints *\
\* Input variables: DomX, LX, ValY, Offset, Maxarg  *\
\* Output variables: X, DomX, LX *\

1)    fc_ne_o_1(X, DomX, LX, ValY, Offset, Maxarg) :-
          Arg is ValY + Offset,          \* compute offset *\
          within(Arg, 1, Maxarg),
          arg(Arg, DomX, (ValY, Flag)),
          (
           if                            \* domain element valid? *\
             Flag \== 0
           then                          \* yes! *\
             Flag = 0,                   \* eliminate value *\
             dec_length(LX, Newlength)
           else                          \* no! *\
              true                        \* finished, yet *\
          ),
          singleton_test(DomX, Newlength, X).


2)    fc_ne_o0_1(X, DomX, ValY, Offset, Maxarg) :-
          within(ValY, 1, Maxarg),
          arg(ValY, DomX, (ValY, Flag)),
          (
           if
             Flag \== 0
           then
             Flag = 0,
             dec_length(LX, Newlength)
           else
             true
          ),
          singleton_test(DomX, Newlength, X).
```

Figure 6.4: Example: Specialized Forward-Checking for Inequality Constraints

The efficiency of the forward-checking use of ordering relations could benefit a lot, if the value of the smallest and the biggest element were stored and maintained separately. Coming back to the example above, in a situation described by

$$\mathrm{Dom}_X = \{3, 4, 5, 6, 7\}, \mathrm{Val}_Y = 2,$$

the algorithm 6.5 would scan the whole domain, instantiating the flags of all the domain elements to zero and finally find out that there is no value left for Val_X compatible with Val_Y, thus produce a failure in the singleton check (step 3 of the algorithm). Having direct access to the minmal value $\mathrm{Min}_X = 3$ would yield an immediate failure, since $3 < 2$ is certainly insatisfiable. This could help us fulfilling one of our goals defined in in chapter 1, i.e. achieving an early detection of failure.
However, maintaining this data leads to a respectable overhead, since every time the domain is manipulated, a check is necessary whether its minimum respectively maximum have changed, and if necessary, the values have to be recomputed. In the prototype implementation of FIDO-II, minimum and maximum are not separately stored, but every time this information is needed

```
\* Input variables:    Dom_X: Domain to be restricted *\
\*                  Length_X: Actual length of Dom_X  *\
\*                     Val_Y: Ground value of Y       *\
\*                         N: Maximal domain size     *\

\* Output variables: *\
\* Val_X: The value X will be bound to if the singleton test is successful *\
\* Dom_X, Length_X: will be modified by the algorithm *\

\* Temporary variable: ArgPtr  *\

   1.   ArgPtr := N  \* choose last domain element as starting point *\

   2.   while (ArgPtr > 0) and Dom_X[ArgPtr]. Value >= Val_Y
          do
            Dom_X[ArgPtr].Flag := 0
            Length_X           := Length_X - 1
            ArgPtr             := ArgPtr - 1
          od
        Goto 3.

   3.   if  Length_X = 0            \* singleton test *\
          then
            STOP FAIL
          else
            if Length_X = 1
              then
                find_single_element(Dom_X, El),
                Val_X := El
              else
                true
            fi
        fi
```

Figure 6.5: A Forward-Checking Algorithm for the < /2 Constraint

[11], it has to be computed by checking the domain.
Finally, I would like to examine how well-suited forward-checking is for ordering constraints. The main pruning effect achievable by using ordering constraints is that some kind of qualitative reasoning over domains can be supported. i.e. without knowing the exact values that can be assigned to the constraint values, it can be possible to exclude values from further considerations if these appear "too small in any case" or "too big in any case".
This qualitative interval reasoning facility is not satisfactorily exploited by forward-checking, since it can be only activated if all up to one variable have been instantiated. In general, that is at a later point during search. This goes obviously against the CLP paradigm of reducing the search space as early as possible. This is why, for the >, <, =< or >= constraints, looking-ahead seems to be a more appropriate consistency technique. Coming back again to our example:
Having $\mathrm{Dom}_X = \{5, 6, 7\}$ and $\mathrm{Dom}_Y = \{2, 3, 4\}$, a looking-ahead use of the constraint $X < Y$ could directly find things going wrong here, whereas forward-checking would have to wait just until one of the variables becomes instantiated.
However, it is true that, since looking-ahead is quite an expensive technique, it can turn out

[11] it will be needed mainly for looking-ahead techniques.

to be better to use forward-checking - but, this must be decided from case to case. By the way, in this context, it seems to be an important feature that the user can change the control behaviour of a program by replacing a **forward** declaration e.g. by a **lookahead** declaration and recompile the program. Thus he can check in a convenient way which strategy works better in the specific application.

6.2.3.6 Local or Global Control

As opposed to e.g. CHIP, FIDO-II uses local **forward** and **lookahead** declarations. Advantages of global control are:

- Elegant formulation of programs.
- Shorter programs.
- Global control is a feature which is assessed positively in many recent approaches [van89a]. By global declarations, it can be organically supported.

I decided to use local control in FIDO-II for the following reasons:

- Using global declarations, a collision semantics has to be defined, if one constraint shall be used under several control regimes.
- Local declarations allow to submit a constraint to different control mechanisms in one program.
- No collisions of declarations for a predicate possible.
- relatively simple implementation.
- The correct declaration to which a constraint is submitted is locally decidable.

6.2.4 Weak Looking-Ahead in FIDO-II

Looking-Ahead [van87b, van87a, Mac77, Lau78] offers a powerful possibility to reduce the number of values that can be assigned to variables of a constraint, even if this constraint is not yet forward-checkable, because there is still more than one domain variable uninstantiated. First, I will shortly repeat the main idea of looking-ahead described in chapter 3.
For every domain variable X_i appearing as an argument of an N-ary constraint $\mathcal{P}$, and for every value within the domain of X_i, it must be checked if there exist value combinations from the domains of the other variables (e.g. at least one admissible value from the domain $Dom_X j$ of each domain variable X_j appearing in $\mathcal{P}$) so that the constraint $\mathcal{P}$ is satisfied. The arguments of $\mathcal{P}$ which are not domain variables must be ground.
Since general looking-ahead is regarded as too expensive for most applications, as I pointed out at the beginning of the chapter, for FIDO-II I implemented a somewhat modified strategy called **Weak Looking Ahead** (**WLA**), that is similar to the one proposed in [dSPRB90]. A definition of the underlying inference rule is given in chapter 3. Since, in this work, I put the stress on forward-checking as a consistency algorithm, I implemented WLA only in an exemplary manner to give an idea of how well-suited it is for a CLP-system based on PROLOG with coroutining. I re-implemented the following constraints in a looking-ahead-like way:

- The equality constraint = /2.
- The < and > constraints without evaluation of their arguments.
- Evaluating versions of < and > working for a class of numeric expressions, i.e. for patterns of the form $X < Y + Z$ resp. $X > Y + Z$.

As we will see in section 6.3, where the implementation of lookahead behaviour is described, there is a principial difficulty in implementing pure looking-ahead using a **delay** mechanism in PROLOG, since we cannot easily formulate the lookahead conditions by **delay** clauses. Particularly, looking-ahead can be used several times for one constraint, and its applicability must be tested very often during computation, i.e. after any change in the domains of the constraints variables. The problem with SEPIA **delay** is that the expressiveness of **delay** statements is limited. Thus, it would turn out to be a bigger problem to enforce the system to remember, wake and resuspend the constraint based on more complex conditions by using **delay** declarations.
In order to solve this problem, weak looking-ahead follows a different strategy:

WLA strategy

- When a WLA constraint call is regarded the very first time, a looking-ahead check is performed on it, hopefully yielding a considerable pruning effect.
- After this, the constraint is not "thrown away", since, in general, it does not have a solved, but a simplified form, yet. Keeping the constraint for solving it later, if additional information is available, is brought about by a forward-checking call of the constraint.

The proceeding described above allows to burden the underlying PROLOG system with the looking-ahead control task. By doing looking-ahead only one time, i.e. quite in the beginning of the constraint solving process, overhead is limited, and a looking-ahead like behaviour can be achieved, which performs a pre-pruning of the search spaces, hopefully making some constraints forward-checkable. Certainly, as it is the case for any heuristic algorithm, there are examples where it works very well and there are examples where using WLA yields no effect at all, even if standard looking-ahead finally would have reduced the search space at a later time during computation. But, this is the price to pay for applying looking-ahead only once for every constraint, and thus for reducing computation overhead and moreover for preserving the completeness of the algorithm.

6.2.4.1 The User's View on Weak Looking-Ahead in FIDO-II

The programmer can declare a predicate call to be submitted to weak looking-ahead control by simply calling it as the first argument of **lookahead** /1. This is exactly the same as described for **forward** declarations in section 6.2.3.1. Since no general WLA algorithm is provided in the current implementation, it will work only for the constraints listed above. If a different constraint is called using a **lookahead** declaration, the FIDO preprocessor will print a warning and the declaration will be ignored. Because of the modularity of constraint redefinitions, however, additional redefinitions and specialized versions of the algorithm can be added to the system by creating a new entry for them into the global constraint description table, which is implemented as a database of PROLOG facts. Thus, the expressiveness of the system and the convenience of programming can be enhanced, if this is desired.

Why There is no General Looking-Ahead in FIDO-II The decision not to implement a general looking-ahead algorithm is based on a couple of considerations.

- First of all, as I said, looking-ahead is an expensive method for ensuring arc-consistency. Using a general algorithm which neither uses knowledge about constraint-specific peculiarities nor exploits domain-specific information, will not counteract this fact, on the contrary, we must expect that the efficiency resulting from this will be relatively poor.
- That is why, in my opinion, it is a better approach to show the usefulness of the method by implementing specialized versions of it for a few constraints whose looking-ahead execution seems promising.
- Another reason is certainly the amount of time and work necessary for implementing it. To me, it was necessary to set limits to the scope of this work, and I decided to set them in the looking-ahead area.
- Therefore, for an exemplary examination of consistency techniques in logic programming, forward-checking seems to be more appropriate since it yields better results than looking-ahead for many applications[12].

However, by explaining the concept of WLA in chapter 3 and by the exemplary implementation of it, I hope to have laid the foundations for an interested reader to understand and implement it for himself without greater problems[13].
In the following, I would like to have a glance at the way WLA is brought about for the constraints mentioned above.
For a looking-ahead use of the equality constraint, however, I would like to refer to section 6.2.5, where that issue is thoroughly discussed.

6.2.4.2 Using Ordering Constraints in a WLA Manner

As I mentioned in section 6.2.3.5, ordering constraints such as $>$ or $<$ are well-suited to looking-ahead methods. Since they incorporate rather reasoning over intervals of values than reasoning about values themselves, they often can be used early and thus, the set of possible values that can be given to the domain variables can be reduced. Their efficient implementation, however, depends on the availability of the minimal and maximal values of the domains, because only in this case, the step from quantitative to a more qualitative reasoning is supported. Let us see now by an example, how WLA works:

An Example Assume the constraint to be solved is

$$X < Y + Z, \text{ where } X, Y, Z \text{ are domain variables,}$$
$$X \text{ ranges over the domain } \{9, 10, 11, 12\},$$
$$Y \text{ ranges over the domain } \{2, 3, 4, 5\} \text{ and}$$
$$Z \text{ ranges over the domain } \{3, 4, 5, 6\}.$$

If this constraint is called by introducing a goal `lookahead`$(X < Y + Z)$, the following happens:

[12] So why should we start again inventing the wheel?
[13] I'm just kidding, there *will* be greater problems, as soon as someone tries that!

First Step For each variable X_i and each value Val_{X_i} within its domain, values from the other variables' domain have to be found which satisfy the constraint. If no such tuple(s) can be found, the value Val_{X_i} is eliminated.
Starting with X in our example, we can find values Val_Y, Val_Z for Y and Z which satisfy the constraint

$$X < \mathrm{Val}_Y + \mathrm{Val}_Z$$

for $X \in \{9, 10\}$. Especially, instead of testing all the values from Dom_Y resp. Dom_Z, we can use the maximum values Max_Y and Max_Z and compute

$$\mathrm{Bound}_X := \mathrm{Max}_Y + \mathrm{Max}_Z,$$

removing from Dom_X all the values Val which are *not* less than Bound_X. Thus, we can eliminate the values 11 and 12 from the domain of X. For Y, in turn, we have to find values Val_X, Val_Z so that

$$Y > \mathrm{Val}_X - \mathrm{Val}_Z$$

is true. This is especially guaranteed for all these values of Dom_Y greater or equal than the difference between the maximum of Dom_X and the minimum of Dom_Z. Thus, Bound_Y can be computed as

$$\mathrm{Bound}_Y := \mathrm{Min}_X - \mathrm{Max}_Z$$

and the condition $Y > \mathrm{Bound}_Y$ has to be tested. As a result of this, all values which are less than 3 can be eliminated from Dom_Y, resulting a new domain $\mathrm{Dom}_Y = \{4, 5\}$. Analogously, Bound_Z results as $\mathrm{Min}_X - \mathrm{Max}_Z$, leaving a new domain $\mathrm{Dom}_Z = \{5, 6\}$. Thus, just by doing one lookahead turn, we restricted the domains to

$$\mathrm{Dom}_X = \{9, 10\}, \mathrm{Dom}_Y = \{4, 5\} \text{ and } \mathrm{Dom}_Z = \{5, 6\}.$$

Second Step In order to keep track of the further execution of the constraint, the WLA routine now starts a call to a forward-checking version of the constraint[14]. As long as more than one of the three variables X, Y and Z remain uninstantiated, nothing will happen. This condition is supervised by the forward-checking control mechanism, which will be thoroughly described in sectiondelays. If, however, only one variable remains uninstantiated, forward-checking will start and the domain of this variable will be restricted by eliminating values which are not compatible with the assignments to the other variables.
The processing described above is reflected in the definition of the looking-ahead version of our example constraint shown in figure 6.6 for the case that X, Y and Z are domain variables.

What is not visible in figure 6.6 and what will be shown in the next chapter is how the initial call `lookahead`$(X < Y + Z)$ is transformed into the call of the specialized constraint `look_ahead_<_d_dpd`(X, Y, Z). Note that in the program above, knowledge about the behaviour of the $<$ constraint is integrated, especially as regards the computation of appropriate bounds for X, Y and Z.

[14] Since we're dealing with built-in constraints here, a specialized **forward** version will be available

```
lookahead_<_d_dpd((&,IdX,LX,CX,X,DX),(&,IdY,LY,CY,Y,DY),(&,IdZ,LZ,CZ,Z,DZ)) :-
     get_min(DX, MinX),          \* Get boundary values *\
     get_max(DY, MaxY),
     get_max(DZ, MaxZ),
     BoundX is MaxY + MaxZ,      \* Compute bounds to be checked *\
     BoundY is MinX - MaxZ,
     BoundZ is MinX - MaxY,
     fc_lt(X, DX, LX, BoundX), \* Remove inconsistent values from DomX *\
     fc_gt(Y, DY, LY, BoundY), \*    "      "          "        "   DomY *\
     fc_gt(Z, DZ, LZ, BoundZ), \*    "      "          "        "   DomZ *\
     forward_<_d_dpd((&,IDX,LX,CX,X,DX),(&,IdY,LY,CY,Y,DY),
                     (&,IdZ,LZ,CZ,Z,DZ)).
```

Figure 6.6: Example: Part of the WLA Algorithm for the > /2 Constraint

6.2.5 Handling Equality

There are some problems concerning equality, which occur as a consequence of the representation of domain variables and which are to be outlined in this section. I would like to describe the realization of domain variable unification in FIDO-II and the user's possibilities to influence it.

6.2.5.1 Unification

The representation of a domain variable as a sixtuple with an explicit identifier (see section 5.2.1) has some consequences for the realization of the general domain variable unification (see chapter 3). This especially concerns unification between two domain variables. Here, we have to make a difference between

1. explicit unification using the = /2 predicate.
2. implicit (clause head) unification internally performed by the PROLOG inference engine calling the unification routine.

These two cases will be regarded in the following.

Explicit Domain Variable Unification Assume there are two domain variables

$$X = (\&, \mathrm{Id}_X, \mathrm{Length}_X, \mathrm{Constraints}_X, \mathrm{Val}_X, \mathrm{Dom}_X),$$
$$Y = (\&, \mathrm{Id}_Y, \mathrm{Length}_Y, \mathrm{Constraints}_Y, \mathrm{Val}_Y, \mathrm{Dom}_Y).$$

These two variables shall be unified by an explicit call to a goal $X = Y$ in a FIDO-II source program. Again, we have to make a distinction between two cases:

1. $\mathrm{Id}_X = \mathrm{Id}_Y$, i.e. both variables have the same domain (strictly speaking, the domain identified by the same identifier, which is a much stronger notion, as we will see).
2. $\mathrm{Id}_X \backslash= \mathrm{Id}_Y$, i.e. the domains of the two variables are denoted by different identifiers in the `define_domain` calls defining the respective domains.

Equal Domain Identifiers Due to its representation of domain variables chosen, in FIDO-II, unification of variables sharing the same domain happens to be quite straightforward. If $\mathrm{Id}_X = \mathrm{Id}_Y$ is true, we can directly map unification to PROLOG unification, i.e. the goal $X = Y$ can be handled by PROLOG's unification routine. Especially the computation of domain intersection is very simple, since it is computed automatically by calling the goal

$$\mathrm{Dom}_X = \mathrm{Dom}_Y.$$

By this, since the domains are really unifiable structures, elements marked invalid in Dom_X will be marked invalid in Dom_Y too, by unifying the validity flag of the domain values.
The only thing that remains to be done is to compute the new domain length Length_X resp. Length_Y[15] and to perform the singleton test, i.e. to check, if the domain contains valid elements after the intersection.

Different Domain Identifiers However, if Id_X and Id_Y are different atoms, things are getting worse. In this case, the two domain variables incorporate different (i.e. not unifiable) syntactical objects, since the different identifiers prevent unification and the internal representation of Dom_X and Dom_Y can be different[16]. Thus, more complicated actions have to be performed.
The main problem in this context results from the underlying language, which is PROLOG. The general domain variable unification algorithm requires that, if two domain variables X and Y are unified, both of them are bound to a new variable ranging over the intersection $\mathrm{Dom} = \mathrm{Dom}_X \bigcap \mathrm{Dom}_Y$. But, we actually can't do this in PROLOG, since a variable cannot change its value, cannot be reassigned.
One possibility of achieving domain variable unification despite that would be to keep track of domains by maintaining an open list where a history of domain states is stored. This is certainly not a good solution, since

- it is expensive in terms of memory,
- domain access causes considerable cost, for the domain itself has to be found, before the values of it can be retrieved, and
- this does not solve the problem of how to guarantee semantic equality of domain variable structure which are syntactically not unifiable.

The latter will turn out to be the hardest, up to now unsolved problem.
Therefore, the desired behaviour has to be simulated somehow. For unifying domain variables of different domains, I propose the following algorithm which is described informally in the following:

Algorithm

1. Traverse Dom_X and Dom_Y, eliminating from both domains each value that does not appear in both of them[17]. As the smart reader will remark, this essentially corresponds

[15] since they must be equal, only one of them has to be recomputed.

[16] Note that it can be different even if the domains are semantically equal, e.g. a domain $d_1 = \{1, 2, 3\}$ and a domain $d_2 = \{3, 2, 1\}$ will be internally represented differently.

[17] Here it can be very useful to exploit information about orders on the domains, for this can drastically decrease search effort.

to some kind of looking-ahead version of the = /2 constraint. We check for each possible value of the one variable if there exists a value satisfying the equality constraint within the domain of the other variable. And indeed, the implementation of this algorithm will use a WLA version of the equality constraint.

2. Recompute the values of the variables Length_X, Length_Y, Constraints_X and Constraints_Y.

3. Ensure that equality between X and Y is guaranteed even if further modifications concerning the domains of the variables are made.

Note that step 3 turns out to be crucial here: in step 1, we have computed the intersection of the two domains Dom_X and Dom_Y at a fixed moment during computation. If we left it that way, we would have done only half a job, because further modifications in one of the domains could be made without being triggered to the other domain. Thus, as we will see, intermediate inconsistencies can appear, and the semantic requirement on equality between X and Y would not be consequently fulfilled. So, how can we solve this problem, or rather, can we solve it, at all? We will see that there is no unique answer to this question.

Ensuring Equality by Unification? If we decide on solving the problem by linking X and Y together simply by unifying their values Val_X and Val_Y, we'd finally guarantee consistency, since, as soon as one of the variables is instantiated to a ground value, due to unification, the same happens to the other one, respectively. The problem with this is that we can produce inconsistent intermediate states of the domains, as I will show in a small example:
Assume that, by the first step of the unification algorithm described above, the domains of X and Y have been restricted to the set $\{1,2,3\}$. Later on, during computation, the value 2 is removed from Dom_X. Since we don't have a survey mechanism to trigger this change to all the domains linked to Dom_X by the unification condition, yet we have a state which is semantically inconsistent to the equality constraint between X and Y. This inconsistency remains until the moment one of the variables is instantiated. Then, PROLOG unification works as a trigger mechanism itself and recovers consistency.
Now, it depends from the desired representation of our solutions, whether this is only silly behaviour or whether it leads to really wrong output. If we always and only consider a vector of variable substitution as a solution of our constraint problem, the phenomenon described above does not matter at all and stays invisible to the user, for in the moment values are assigned to the variables, consistency is satisfied again. In this case, we merely use efficiency. In our example above, we could have removed the value 2 from Dom_Y earlier, thus pruning the search tree a bit more.
But, considering existing CLP systems, such as [JL87], by the authors emphasis is laid on the fact that these systems facilitate an implicit representation of solutions, such as characterizing a solution as the set of remaining domain elements. If we want to provide such an implicit representation, the behaviour described above will lead to an error, as the following example 6.7 describes.

Handling explicit unification as described above first reduces the domains of X and Y to the set $\{1,2,3\}$ (steps (1) and (2) in figure 6.7). The \= /2 constraint actualizes Dom_X to $\{1,3\}$, since it is **forward** executed. An implicit solution of this program could be gained by simply printing the resulting variable domains together with the actually delayed constraints. Since X = Y is not really delayed, but only known to the underlying PROLOG system as having the same value, an explicit solution of this program would be

```
erroneous_example(X, Y) :-
    define_domain(d1, [X], [1, 2, 3, 4]),    \* (1) *\
    define_domain(d2, [Y], [0, 1, 2, 3]),    \* (2) *\
    X = Y,                                   \* (3) *\
    forward(X \= 2).                         \* (4) *\
```

Figure 6.7: Example: A FIDO Program Delivering an Inconsistent Implicit Solution

```
correct_example(X, Y) :-
    define_domain(d1, [X], [1, 2, 3, 4]),    \* (1) *\
    define_domain(d2, [Y], [0, 1, 2, 3]),    \* (2) *\
    X = Y,                                   \* (3) *\
    forward(X \= 2),                         \* (4) *\
    instantiate([X, Y]).                     \* (5) *\
```

Figure 6.8: Turning Things Right by Enforcing Explicit Solutions

- $X \in \{1,3\}, Y \in \{1,2,3\}$
- The empty set $\{\}$ of delayed goals.

This is actually not the semantics intended by the standard interpretation of the program!

A simple possibility to remedy this from the user's view is simply to swap the program lines marked with (3) and (4), since then WLA would directly remove the value 2 from Dom_Y. Another possibility to yield a correct result is shown in figure 6.8. By instantiating the variables using the FIDO-II library predicate `instantiate / 1`, which binds a list of domain variables to admissible values from their domains, consistency is forced. On backtracking, all solutions will be created[18]. A possibility of solving this dilemma is pointed out in the following.

Using Forward-Checking Execution to Ensure Equality? Having unification of the variable values as a first step, a second thing one could do is calling a forward-checking version of the equality constraint in order to ensure consistency[19]. The advantage of doing that is that the goal

```
forward(X = Y),
```

which will be called after the first and the second step of Algorithm 6.2.5.1, is supervised by the `delay` mechanism of the system. Coming back to our example shown in figure 6.7, the situation is the following, if an implicit solution is desired: We have

- The variable domains $\mathrm{Dom}_X = \{1,3\}$, $\mathrm{Dom}_Y = \{1,2,3\}$
- A set of delayed goals $\{X = Y\}$.

[18] Note, however, that in systems managing infinite domains, explicit enumeration is not satisfatory. There, the possibility of an implicit description must be given.

[19] For a detailed description of forward-checking, see section 6.2.3.

Thus, by combining these two pieces of information, the implicit soundness of this solution can be made explicit. This can be done by applying algorithm 6.2.5.1 (or I could also say a WLA call) to each equality constraint still in the delayed-goals list. In the above example, by such a call, the value 2 would be removed from the domain Dom_Y, delivering a correct result.

How to Do it in FIDO-II Since we have finite, discrete domains in FIDO-II, representing solutions implicitly is not such an important feature. That is why I implemented step 3 of algorithm 6.2.5.1 by unifying the value variables Val_X and Val_Y of X and Y and by calling the forward-checking version in order to ensure final correctness of the examples. But I have not taken the pain here to implement the last transformation which combines the information from the domains of not instantiated variables with the goals remaining in the delay-list.

Conclusion Since we can't reassign values to PROLOG variables, a modification of explicit domain variable unification is necessary. It is correct for programs finally instantiating the variables, e.g. for all programs working in a generate & test manner, whereas it sometimes yields erroneous results if an implicit solution is desired, i.e. if variable instantiations are avoided. However, I pointed out a general way how to remedy this.

Implicit Unification of Domain Variables Up to the current SEPIA version, it has not been possible to control and modify unification. Thus, implicit unification of domain variables of different domains cannot be handled by FIDO-II. An example of such a (forbidden) situation is shown in figure 6.9.

```
erroneous_example2(X, Y) :-
    define_domain(d1, [X], [1, 2, 3]),
    define_domain(d2, [Y], [1, 2, 3]),
    erroneous_example2(Y, X).
```

Figure 6.9: Example: Erroneous Implicit Unification

What is striking with this example is the fact that, although X and Y range over (semantically) identical domains, their implicit unification fails. By the recursive call with swapped arguments, PROLOG tries to apply its standard unification routine to the variables X and Y - which does not work due to the syntactically different structure of the variables. Thus the call to `erroneous_example2(X, Y)` fails instead of running in an infinite loop, as one would expect.
However, in the latest SEPIA version 3.0.16 [SEP91], it will be possible to check and redefine unification in an appropriate manner by combining the SEPIA event handling mechanism and the new concept of **meta-terms**. Unfortunately, that came too late to be considered in this work. This new feature and the effects it could have had on the implementation of FIDO-II are outlined in section 5.3.
Thus, in the current FIDO-II implementation, the user has to take care of not applying clause head unification to domain variables. As we've seen by example 6.9, that kind of unification will always fail. Even the clause head unification between a domain variable and a constant cannot be performed, since PROLOG's unification routine actually can't interpret the domain variable structure in a correct way. Certainly, by a more sophisticated static program analysis

and horizontal program transformation, some of these cases could be recognized and, instead of produce a failure that will be quite miraculous to the programmer, at least a warning could be created concerning implicit unification involving a domain variable. However, this remains a general problem and a lack of expressiveness of the approach.

6.2.5.2 How the User Can Influence Unification

If the standard = /2 predicate is used in order to express unification involving domain variables, FIDO-II will use the algorithm described in the previous paragraph. It will redefine the equality constraint performing a case distinction in order to handle different argument patterns.
However, if the user is content with a simplified, but more efficient version of = /2, which does not support explicit unification between domain variables ranging over different domains (i.e. which fails in this case), he can use a special FIDO-II equality operation :=: /2, denoting a *restricted domain variable unification* operator. The behaviour of :=: /2 subsumes standard PROLOG unification. Besides, it can handle explicit unification between

- a domain variable and a constant,
- a domain variable and a normal variable,
- two domain variables whose domains share the same identifiers.

It fails, however, if it is applied to domain variables of different domains. Thus, if not needed, the overhead caused by domain variable unification can be decreased. Moreover, the preprocessor is able to check whether a predicate is called with domain variable arguments. If this is not the case for unification in a program, it does not have to be redefined, but can be handled by the fast PROLOG unification routine.

6.3 Using `delay` Mechanisms

In this chapter I will descibe how an advanced control behaviour is achieved in FIDO-II. Constraints provided with a `forward` declaration will be redefined using the SEPIA `delay` mechanism in order to simulate the forward-checking inference rule. `delay` conditions will be formulated on the redefined constraints, thus deferring the execution until the variables of the constraints are sufficiently instantiated.
After a few words about coroutining in PROLOG, I should like to introduce SEPIA's `delay` features by an example and show how forward-checking can be implemented on top of such a mechanism.

6.3.1 Approaches Towards Coroutining

The idea of providing a coroutining mechanism for logic programming originates from the effort aimed at making PROLOG control strategies more efficient. This idea has been given expression to several times since the end of the seventies. Pioneer work in this direction has been done by [CM79] with IC-PROLOG, by Colmerauer [Col82, CKv83, Col87a] introducing the *geler* predicate in PROLOG-II, by [Car87] introducing *dif* and *freeze* and by the conceptual and practical work of Naish [Nai82](MU-PROLOG). Also the work of [DL84] (METALOG) and of [GL80] should be mentioned in this context. During the last years, almost all important

PROLOG systems followed offering a coroutining mechanism.
The idea which is common to all these approaches is the following:
Sometimes, it may be desired that not all input arguments of a predicate should be instantiated when the predicate is called. In the case of being called with some uninstantiated input arguments, the operational behaviour of the predicate should be modified:
it shall not fail, but its execution shall be deferred until, later on during computation, values are assigned to the respective variables.
Thus, from the view of *constraint solving*, a constraint net is built, which can be propagated later on by giving values to some variables. That idea is pursued in this work. Generally, coroutining gives the possibility to obtain partial evaluation in logic programming and thus, to enhance the scope and the efficiency of logic programming languages, while preserving their logic part. Its main effect is to enable programs written in a *generate & test* style to simulate standard backtracking search. Thus, the elegance of generate & test and the (relative) efficiency of standard backtracking can be combined. This is achieved by adding control information to the program. A fundamental paper dealing with this issue is [Nai85].
However, a mere use of coroutining aimed at achieving a better control strategy for logic programming languages is nothing more than cosmetics to generate & test programs, since it only yields passive pruning and does not improve the behaviour of standard backtracking at all. What we are doing here is making use of coroutining in order to implement a forward-checking behaviour. This opens up to us a higher level of efficiency, because it allows constraints to reduce the search space actively.

6.3.2 SEPIA `delay` Declarations

As many PROLOG systems do (e.g. MU-PROLOG [Nai82], IC-PROLOG [CM79], NU-PROLO([TZ88]), a coroutining facility is integrated in the advanced feature PROLOG system SEPIA [MAC+89, SEP90, SEP91]: the SEPIA **`delay`** mechanism. That feature is to be used in the practical work accompanying this work. PROLOG is extended by the possibility of formulating **`delay`** declarations on predicates. That way, execution conditions (or rather: delay conditions) can be expressed, thus directing control in the way desired. Example 6.10 shows what such a **`delay`** declaration on a predicate call can look like. The example shows the implementation of a symmetrical verison of the real multiplication predicate */3. Thus, sym_*/3 can be used in order to ask the queries, shown in figure 6.11. Especially interesting is the last case. Since the first and the last argument of the goal are uninstantiated, the call to **`sym_*`**$(X, 5, Y)$ remains delayed. This gives us the possibility to express implicit solutions for problems. That can be very convenient, as example 6.12 shows. Assume the predicate **`admissible`** / 3 checks whether a combination of three values X, Y and Z is admissible for some purpose. The semantics of **`admissible`** is obvious. By the use of coroutining, implicit solutions can be given. i.e. the query

```
?- admissible(6, Y, 3)
```

yields the following answer:

```
Y = _d58        Delayed constraints:
                6 > _d58
                _d58 > 3
yes.
[sepia]:
```

```
\* implementation of the symmetrical multiplication operator based on delay *\

    sym_*(X, Y, Z) :-
         nonground(Y),
         !,
         Y is X / Z.

    sym_*(X, Y, Z) :-
         nonground(Z),
         !,
         Z is X / Y.

    sym_*(X, Y, Z) :-
         X is Y * Z.

    delay    sym_*(X, Y, Z)
       if    nonground(X),
             nonground(Y).

    delay    sym_*(X, Y, Z)
       if    nonground(X),
             nonground(Z).

    delay    sym_*(X, Y, Z)
       if    nonground(Y),
             nonground(Z).
```

Figure 6.10: Example: Use of a SEPIA `delay Declaration`

```
[sepia]: sym_*(6, 3, X).

X = 2.0
yes.
[sepia]: sym_*(Z, 7, 3).

Z = 21
yes.
[sepia]: sym_*(4, X, 12).

X = 00.333333
yes.
[sepia]: sym_*(X, 5, Y).

X = _d74
Y = _d66          Delayed constraints:
                  sym_*(_d74, 5, _d66)
```

Figure 6.11: Symmetrical Multiplication Predicate `sym_*`/3

Thus, the implicit solution condition for variable Y (Y must be between Z and X) can be fomulated even without giving an explicit numerical solution concerning the value(s) that can be assigned to Y.

```
admissible(X, Y, Z) :-
    X > Y,
    Y > Z,
    X > Z.


delay    admissible(X, Y, Z)
   if    nonground(X),
         nonground(Y).

delay    admissible(X, Y, Z)
   if    nonground(X),
         nonground(Z).

delay    admissible(X, Y, Z)
   if    nonground(Y),
         nonground(Z).
```

Figure 6.12: Yielding Implicit Solutions by Delaying Goals

Let me come back now to example 6.10. Before each call to the goal **sym_*** / 3, it is tested whether the delay condition still holds. If it does not, the goal is executed like a normal PROLOG goal, else the goal is delayed, i.e. it is pushed on a *delay stack*. Each time a variable appearing in the delayed goal is bound or unbound, it is tested again if the delay condition holds. If not, the goal will be popped from the delay stack (it is *resumed*) and it will be made current goal.

Peculiarities of the SEPIA Resuming Strategy In this context, I would like to make a short remark on an interesting effect of the SEPIA behaviour described above. Figure 6.13 shows an example situation:
Assume there is a goal

$$A :- A_1, \ldots, A_N.$$

By the execution of the current goal, which shall be A_M, $1 \leq M \leq N$, a variable, say X_1 is instantiated, causing a number of delayed goals

$$C_{11}, \ldots, C_{1J}, \ldots, C_{1K1}$$

to be resumed[20]. Due to SEPIA resume strategy, the first of the woken goals, i.e. C_{11}, becomes new current goal. During the execution of a C_{1K}, however, another variable, X_2, can be instantiated, again causing constraints C_{2L} to be resumed, and so on. Thus, a *resuming cascade* can arise resulting from the initial variable assignment to X_1. This is shown graphically in figure 6.13. The problem resulting from this is that the most recently woken goals become current goals. There is a depth first search behaviour resulting from that, starting from a

[20]it is obvious that by instantiating one variable, several constraints can be resumed since a variable can appear in moe than one constraint.

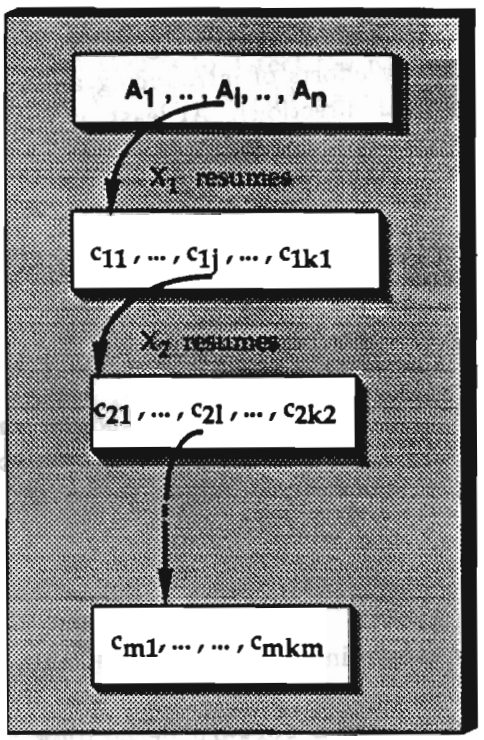

Figure 6.13: A SEPIA Resuming Cascade

variable instantiation (which can be not guaranteed to be right itself, i.e. which can turn out to be wrong later on, causing backtracking), thus potentially creating lots of new goals. By this depth-first resuming strategy, the number of choice points is drastically increased, so that also backtracking probability is increased. It is well-known that, for infinite search trees, depth first is not a safe method for finding a solution, since it resembles shooting into the night, hoping to hit something edible.[21]

In my opinion, a better behaviour would be to try first the goals resumed first (i.e. a FIFO organization of resuming). Thus, backtracking frequency as well as the number of goals to consider could be decreased in many cases, the probability of discovering a failure early would increase. In our actual example 6.13, this would mean to continue from goal C_{1J} with goal C_{1J+1}, queuing the new resumed goals C_{21}, ..., C_{2K2} behind C_{1K1}. That way, a breadth-first component could be introduced into the search. In my opinion, this would be a more appropriate method for straightforward "left-to-right-with-backtracking" algorithms such as N queens and other discrete combinatorical problems discussed in this work (see chapter7).

I talked about that issue to some people from ECRC. They had their doubts whether changing the goal-selection-strategy from depth-first to breadth-first could yield positive results. Their main arguments were that

- the idea of introducing a breadth-first method for subgoal selection in the case of resume cascades would not correspond to the depth-first goal-selection philosophy of PROLOG[22].
- nobody had ever complained about this up to now.

[21] The risc of starving is said to be quite high, that way!

[22] How puristic people can be if it comes to avoiding additional work!

It is certainly true that both strategies discussed are heuristics. That implies that for each of them, examples can be found that work wonderful, whereas other examples yield pathological results. However, the idea of introducing a breadth-first component into PROLOG seems interesting to me (many efforts of introducing and/or parallelism into PROLOG [WRCS87, van89b] aim at a similar direction). At least, it could be worth trying it and watching the results.

6.4 Consistency Techniques in FIDO-II

6.4.1 Realizing Forward-Checking via a `delay` Mechanism

In this section, I describe the way a forward-checking control behaviour is obtained in FIDO-II. This is done by using SEPIA **`delay`** declarations on specialized constraints in order to defer the execution of a goal which is not sufficiently instantiated.

6.4.1.1 Built-in Constraints

Here, using the $=\backslash=$ constraint as an example, the realization of control for built-in constraints will be shown.
Assume the user formulates a **`forward`** declaration on $=\ \backslash\ =$, e.g. as happens in the N queens program in figure 7.1. What the preprocessor has to do is to replace each call to **`forward`**$(=\backslash=/2)$ by a call to **`for_nne`** /2, which is the built-in **`forward`** redefinition of $=\backslash=/2$.
The **`forward`** redefinitions of the built-in constraints are predefined and can be accessed (i.e. loaded) by the system. That is a difference to the way the redefinitions of user-defined constraints are provided. The latter ones are dynamically created by a code generating module during precompilation.
The predefined constraints, however, are implemented more efficiently, loading only the definitions which are really needed.[23] Let us have a glance now what the redefinitions are implemented like.
The transformation of the FIDO constraints into executable PROLOG code is performed in three steps:

1. Instance Test All constraints pass a normalizer during preprocessing. There they are transformed into a normal form. Numerical constraints, as e.g. $=\backslash=/2$ are reshaped into a set of new constraints, each of them being an instance of one of the following patterns:

- $X =\backslash= Y$
- $X =\backslash= Y + Z$
- $X =\backslash= Y * Z$
- $X =\backslash= Y + N * Z$,

[23] One could achieve that using a dynamic generator, too, but this is not performed by the prototype preprocessor.

where X, Y, Z are simple (see definition 12).
This is done because specialized redefinitions are implemented for these patterns which allow efficient processing. Thus, the first step is to detect the pattern corresponding to the actual constraint call. e.g. the expression $X =\backslash= Y$ would match the first of the above cases. Thus, a clause

```
(1)    for_nne(X, Y) :-
          for_nne0(X, Y).
```

is added to the source code. We cannot simply replace `for_nne` /2 by `for_nne0` /2, because it is still possible that the source program contains a goal using $=\backslash=$ /2 in a forward-checking way, but with a different argument pattern, e.g. a call to

$$\texttt{forward}(A =\backslash= B + C).$$

In this case, another clause,

```
(2)    for_nne(X, Y) :-
          instance(Y, X1 + Y1),
          !,
          for_nne1(X, Y).
```

must be added, where `for_nne1 / 2` handles the sum in the second constraint argument. By the way, in order to achieve the desired procedural behaviour, clause (2) must be added before clause (1) in the destination file.
These pattern-dependent case distinctions are created dynamically. Thus, only necessary cases are taken into account, reducing the *fan-out* and avoiding redundant search effort and nondeterminism.

2. Domain Variable Test Since it is not clear at compile time, which constraint arguments will be domain variables at run-time and which will not, we have to check that by an explicit case distinction during run-time. For each argument combination *domain variable* $\bowtie$ *normal variable*, a specialized version of the constraint will be called. Note that this leads to an explosion of possible cases for constraints having a bigger number of arguments, since for an N-ary constraint, there exist 2^N possible combinations, for each argument can be a domain variable or a normal variable. However, from a practical point of view, this is not too bad, because most interesting constraints do not have more than three or four arguments. Many of them are even binary constraints.
That potential combinatorical explosion is the reason, why, for user-defined constraints, the maximal number of arguments is restricted to 5 in the current FIDO-II implementation. Figure 6.14 shows the case distinctions for the for_nne0(X, Y) constraint.

3. Specialized Redefinitions At the third level of redefinition, the `delay` mechanism becomes visible. Following the definition of the FCIR, a constraint can "fire" in a forward-checking manner if all except one of its domain variable arguments are ground, and if all other arguments are ground, too.
For an n-ary constraint C with $k \leq n$ variable arguments, this can be formulated using $\binom{k}{2}$

```
for_nne0(X, Y) :-        \* both X and Y are domain variables *\
    X = (&,_,_,_,_,_),
    Y = (&,_,_,_,_,_),
    !,
    for_nne_dd(X, Y).

for_nne0(X, Y) :-        \* only X is a domain variable *\
    X = (&,_,_,_,_,_),
    !,
    for_nne_dn(X, Y).

for_nne0(X, Y) :-        \* only Y is a domain variable *\
    Y = (&,_,_,_,_,_),
    !,
    for_nne_dn(Y, X).

for_nne0(X, Y) :-        \* neither X nor Y are domain variables *\
    X ~= Y.       \* --> use SEPIA sound inequality constraint *\
```

Figure 6.14: Case Distinctions for the `for_nne0` Constraint Redefinition

delay declarations since C must be delayed if any two variables X and Y out of the k arguments of C are nonground. In our example, we need only $\binom{2}{2} = 1$ **delay** declaration, which expresses that the execution of the constraint shall be delayed if both its arguments are nonground. Figures 6.15a. and b. show the specialized constraints for two cases:
In figure 6.15a., X and Y are domain variables, whereas in figure 6.15b., only X is a domain variable. Each definition of specialized constraints contains again a case distinction, because the "non-delay" condition is nondeterministic. If the specialized constraint can fire, we only know that not both its arguments are nonground, or, in other words, that at most one argument is nonground. To find out which on it is, we need to consider $k+1$ cases for a constraint with k variable arguments: it can be each of the k variables, or there is no unbound variable remaining in the call, which corresponds to case number $k+1$.
In our example, for each of the specialized redefinitions, we have three clauses reflecting the possible case distinctions.
In the bodies of the definitions, a forward-checking algorithm is called in order to restrict the domain of the remaining nonground variable. The general call to forward_ne /5 will be replaced by a call to a specialized forward-checking algorithm later on during preprocessing, when more information about the domains involved is available (This information is gathered from the static program analysis). For more information about specialized constraints see section 6.2.3.

6.4.1.2 User-Defined Constraints

In principle, the way things go here is the same as proposed for built-in constraints in the last paragraph. However, instead of *loading* predefined redefinitions, they are *generated* dynamically by a code generator module following the three-level description given in paragraph 6.4.1.1.
In order to restrict the combinatorical explosion induced by the domain variable case distinction (level 2), the number of arguments of user-defined constraints is limited to 5 in the current FIDO-II implementation.

```
a.   for_nne_dd((&,Id,LX,CX,X,DX), (&,_,_,_,Y,_)) :-
         nonground(X),
         !,
         forward_ne(X, Id, DX, LX, Y).

     for_nne_dd((&,_,_,_,X,_), (&,Id,LY,_,Y,DY)) :-
         nonground(Y),
         !,
         forward_ne(Y, Id, DY, LY, X).

     for_nne_dd((&,_,_,_,X,_), (&,_,_,_,Y,_)) :-
         !,
         X =\= Y.

     delay  for_nne_dd((&,_,_,_,X,_),(&,_,_,_,Y,_))
        if  nonground(X),
            nonground(Y).


b.   for_nne_dn((&,_,_,_,X,_), Y) :-
         nonground(Y),
         !,
         X =\= Y.

     for_nne_dn((&,Id,LX,CX,X,DX),Y) :-
         nonground(X),
         !,
         forward_ne(X, Id, DX, LX, Y).

     delay  for_nne_dn((&,_,_,_,X,_),Y)
        if  nonground(X),
            nonground(Y).
```

Figure 6.15: Definition of Specialized $=\backslash=/2$ Constraint

6.4.1.3 Evaluation

In the following, a short assessment of achieving forward-checking control by a **delay** mechanism is given.
I myself feel a little uneasy about the way of obtaining forward-checking behaviour presented for built-in and user-defined constraints. The main point for that is the distinction between domain and "normal" variables leading to a variety of cases that must be handled on PROLOG level. This is awkward, since the distinction is not a logical one, since *each* variable is a domain variable, even if the domain happens to be the Herbrand universe, as it is the case for PROLOG logic variables. The problems resulting from the duality of variable representation in FIDO-II have been outlined in section 6.2.5.1. That way, the system could greatly benefit from a closer integration of *domain variables* and *normal variables* in a common framework. The theoretic work of Jaffar, Lassez[JL87, JM87] and vanHentenryck[van89a] shows that such an integration is possible on a conceptual layer. I think that the realization of an integration of domain variables can be merely achieved on a lower system level, i.e. in a deeper integrated system, since it actually requires the modification of the unification routine. Implementing such a concept on a higher level has always something unnatural. Looking at the way FIDO-II does it seems to justify that statement:
The current state of the FIDO-art implies a three level processing of each n-ary constraint C:

- On the first level, there are (maximally) four pattern distinctions
- On the second level, there have 2^n domain variable case distinctions to be made, and, finally
- On the third level, for each case distinction made before, there are maximally $(n+1)$ cases to be considered, plus $\binom{n}{2}$ **delay** declarations to be made

Thus, the all over amount of possible case definitions is

$$4 * 2^n * (\binom{n}{2} + n + 1),$$

which implies exponential space requirements.
As I said before, it is true that this is *not* acceptable in general, but it is acceptable if it is guaranteed (either by convention or by normalization) that the arity of constraints is kept relatively small.
As a matter of fact, for most problems, the performance of FIDO-II is really tenable, as we can see for several applications in chapter 7. This partially justifies the approach chosen here. But if we intend to make the system comparable with high-performance CLP systems such as CHIP, a deeper integration of control seems one of the most urgent things to do, especially as regards the consistency techniques manipulating domain variables. As a conclusion, we can say that, once more, the domain variable representation turns out to be the bottleneck of FIDO-II.

6.4.2 Implementing Looking-Ahead

The implementation of looking-ahead does not induce any new concepts concerning the use of **delay** mechanisms. Rather, it is implemented by forward-checking and standard PROLOG. Looking-ahead could be implemented by making use of recursive calls to the forward-checking algorithm. I discussed the implementation of forward-checking using **delay** declarations in section 6.4.1. The realization of weak looking-ahead has been described in chapter 6.2.4. Concerning **delay** aspects, it is implemented by forward-checking, too. Thus, I would like to refer the interested reader to these chapters instead of repeating that stuff here.

6.5 Choice Methods in FIDO-II

6.5.1 Motivation

By using consistency techniques such as forward-checking, it is possible to obtain an a priori reduction of the search space. Thus, one of the goals postulated in chapter 1 has been achieved. Also the number of choices that have to be made in order to find a solution of a problem can be reduced by those techniques.
However, since the nature of most CSPs is $\mathcal{NP}$-complete, we cannot entirely avoid making choices. The way choices are made is crucial for the overall efficiency of problem solving, since choices introduce nondeterminism and contribute that way to the phenomenon known as combinatorical search space explosion. Wrong or awkward choices will lead to failure, thus invoke backtracking with all its negative consequences.

Thus, CLP algorithms should be regarded as combining the processes of constraint solving and of making choices in a clever manner [van89a], since both activities determine the overall system performance. Making choices in CLP means instantiating variables, i.e. assuming that a variable has a distinct value. That additional information can be used by the constraint solver in order to restrict the domains of other variables. In this chapter, I would like to show how first-fail heuristics can be used in order to select the variables to be instantiated in a smart way[24].

6.5.2 First-Fail Heuristics

So what is a first-fail heuristics? What we want is to choose the variable to be instantiated next as cleverly as possible. That variable shall be selected whose instantiation will make failure obvious as early as possible. Thus, the `first-fail` principle [HE80] says that
"To succeed, try first where you are most likely to fail."
In terms of CSPs, we could reformulate this in
"To succeed, instantiate the most constrained variable first". In standard logic programming languages, finding out which is "the most constrained variable" is a problem which is not easy to solve. The FIDO domain concept, however, gives a convenient possibility of achieving that goal, as we will see in the following.

6.5.2.1 Realizing First-Fail Heuristics

There are basically two ways of obtaining a first-fail behaviour, which can be used independently or can be combined. The methods are called

- first-fail on domain size (written as FF_{ds}) and
- first-fail on constraint number (written as FF_{cn}).

First-fail on Domain Size The strategy here is to select the variable with the smallest domain to be instantiated next. This embodies a first-fail heuristics for two reasons:

1. Since each domain value corresponds to a choice point, the number of potential choice points induced by a variable instantiation is minimized.
2. Since a constraint fails iff there are no values left within the domain of one of its variables, choosing the smallest domain promises discovering failure as early as possible.

First-fail on Constraint Number The motivation behind that heuristics is to instantiate first the variable that appears in the greatest number of constraints. Having instantiated that variable, say X, each constraint containing X can use the information delivered by the new variable binding. Some of them may become able to prune the domains of other variables, thus, again leading to an earlier detection of failures.

[24] There are other possibilities of optimizing choices, e.g. the use of *constraints as choices* or techniques such as *domain splitting* (see [van89a]). Those, however, are not taken into consideration in FIDO-II.

Combining Both Methods Using only one of the methods introduced above has a disadvantage. At the beginning of computation, when the domains have not been restricted a lot, and when no constraints can "fire", it is often the case that a choice has to be made[25]. If we use only FF_{ds}, there will be probably many variables having the same domain size. Thus, the quality of choices based upon FF_{ds} tends to be quite poor in the beginning of constraint propagation.
To overcome that, a combination of both methods can make sense, according to the following:

1. Compute a list of domain variables with minimal domain sizes.
2. From that list, choose the variable that appears in the biggest number of constraints. If there is more than one variable, choose any of them.

Combining these two heuristics to a new one, we can expect that, even relatively early in the constraint solving process, good choices are made.

6.5.2.2 FIRST-FAIL in FIDO-II

Introducing a first-fail heuristics requires keeping track of various statistical data concerning domain variables, such as the number of constraints and the domain length. Since, in FIDO-II, due to the PROLOG variable concept, it is not possible to change the value of a variable (e.g. decrementing the length of a domain by one), storing and accessing that data causes some overhead to the system.

- For each value removed from the domain, the length variable X_{Length} of a domain variable X has to be decremented.
- For each constraint the variable X appears in, the variable $X_{Constraints}$ must be incremented and later on, if the constraint execution has been finished, be decremented again.

In FIDO-II, the actual domain length is used for the singleton test. Thus, there is only a small overhead (finding the variable with the smallest domain) in realizing an FF_{ds} heuristics.
However, I didn't implement FF_{cn}, mainly for reasons of time. The internal representation of FIDO-II domain variables (see section 5.2.1) provides a variable for storing the constraint number for each variable. Thus, integrating an extension maintaining that variable can be done quite easily if it is desired. For every call to a constraint redefinition calling a consistency algorithm, the following has to be done:

- Before the call, for each variable X appearing in the constraint, an instruction incrementing the $X_{Constraints}$ variable has to be inserted.
- Directly behind the call, an instruction decrementing $X_{constraints}$ must be added.
- an additional instantiation predicate realizing the FF_{cn} or the combined instantiation strategy has to be provided.

For using FF_{ds} in FIDO-II, a new library predicate has been provided:

[25] especially, if we use forward-checking which can be done only if all up to one argument of the constraint are bound.

`instantiate_dl / 1`

accepts a list of domain variables as input argument and, spoken procedurally, instantiates each variable of the list according to a FF_{ds} heuristics. Thus, by simply using `instantiate_dl` instead of `instantiate`, normal instantiation order can be changed into first-fail instantiation order. Which one works better for a specific application, can be checked very easily that way.

6.5.2.3 Results for First-Fail Heuristics

The performance of many problems can be drastically improved by using first-fail heuristics. e.g. the 32 queens problem can be solved within ca. 3 seconds using a first-fail heuristics, whereas without first-fail, even after one hour no solution has been found (see tables 7.1, 7.2). For other problems, using first-fail doesn't show a visible effect, see e.g. table 7.3 for the five houses problem. That should not surprise, since it actually is an heuristic with strong points and shortcomings. Exact run-time data is displayed in chapter 7.

6.6 The FIDO-II Preprocessor

In this chapter, we'll have a glance at the internal structure and the behaviour of the FIDO-II preprocessor. After giving a brief description of the preprocessor interfaces, I will present the single components (static structure) and the internal data and control flow (dynamic structure) in the preprocessor.

6.6.1 The Preprocessor as a Black Box

The main task of the FIDO-II preprocessor is to transform a program written in PROLOG with some extensions into an executable PROLOG program. The resulting program can handle domain variables and it simulates advanced control mechanisms according to control instructions in the source program. In short, FIDO-II performs a *horizontal source-to-source transformation.*
Figure 6.16 shows a view on the preprocessor as a black box. The user interface to FIDO-II allows the declaration of domains and domain variables, and the declaration of advanced control strategies for particular constraints. Finally, first-fail heuristics can be chosen in order to achieve a good instantiation order for variables.

The back-end of the preprocessor, i.e. the system interface, consists of a set of library predicates taking on the task of generating domains and domain variables, predefined constraint redefinitions with `delay` declarations and first-fail library predicates. The generation includes "`?- compile ..`" instructions for loading files as well as PROLOG code for constraint redefinition. The source program is read from an input file, the resulting PROLOG program is written to an output file. Both source and destination file can be specified using the goal

```
?- fido(InFile, OutFile).
```

This starts the preprocessor with the file denoted by InFile. The ouput file, OutFile, is created and the source-to-source transformation is performed. FIDO automatically loads the redefinition files needed and compiles the destination file. Thus, after the successful finish of FIDO-II, the desired query can be put to the system.

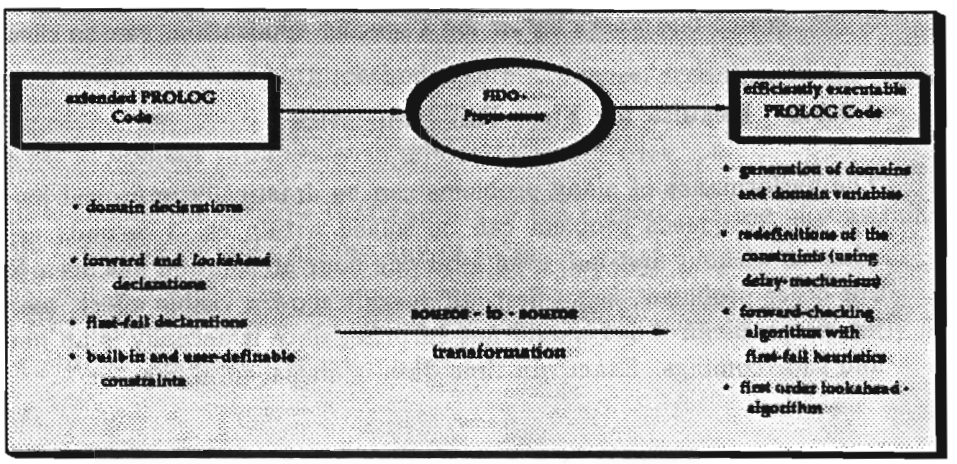

Figure 6.16: The FIDO-II Preprocessor as a Black Box

6.6.2 Static Structure of the FIDO-II Preprocessor

Let us now have a look at the components of FIDO-II and the tasks they fulfil. Figure 6.17 shows the static structure of the FIDO-II preprocessor. I would like to describe in more detail the most important features of the components shown in that figure.

6.6.2.1 The I/O Module

This module is responsible for file in- and output (don't be surprised!). It reads the FIDO source file and writes back the output to the destination file denoted by the programmer. Tightly coupled to I/O is a *classifier* component which performs a first scan of the input file. For reasons of efficiency, this is done while reading the source file. Thus, physically, the classifier is nested into the I/O module. From the logical point of view, however, it can be regarded as an independent module, as illustrated in figure 6.26.

The Classifier Basically, the classifier performs two tasks. The first one is to split the source code in two portions,

- a normal PROLOG part and
- a part containing FIDO extensions.

By that, it can be easily detected whether the source code contains FIDO extensions, i.e. domain and consistency declarations. If not, the preprocessor will write the original code into the destination file since the input program is a standard PROLOG program. The second task of the classifier within the I/O module is to initialize an internal representation of the source program. I denote that representation *callsgraph*, because it is a graph that represents "who calls whom ":

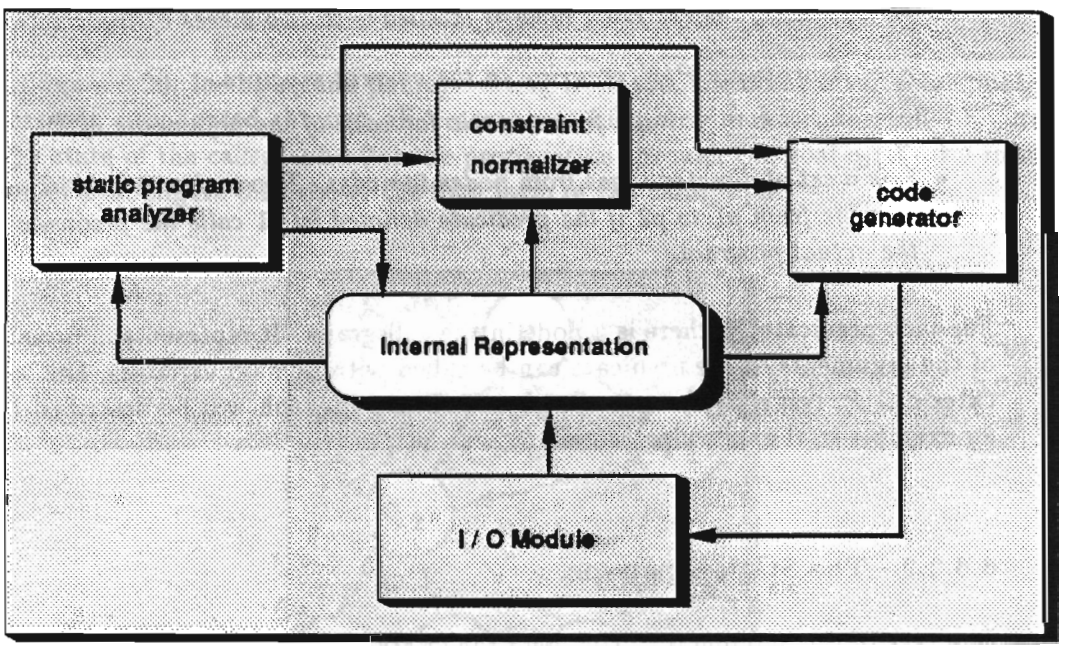

Figure 6.17: Static Structure of the FIDO-II preprocessor

```
p(X1, Y, X2) :-
    define_domain(xyz, [X1, X2], [1, 2]),
    q1(X1, Y),
    q2(Y, X2).

q1(X, Y) :-
    q2(Y, X).

q2(X, Y) :-
    X =< Y,
    Y \= 1.
q2(X, Y) :-
    X = Y.
```

Figure 6.18: An Example Program

Definition 15 (Callsgraph) *A callsgraph is a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where*

- *$\mathcal{N}$ is a set of nodes. Each node is labelled with two pieces of information:*
  - *A predicate P/N occuring in the current program.*
  - *A list containing a free variable for each of the arguments of P.*
- *$\mathcal{E}$ is a set of directed edges from nodes to nodes. Two nodes p1 and p2 are connected with an edge from p1 to p2 if the predicate denoted by p1 calls the predicate denoted by p2 in the actual program.*

For each predicate P, there is a node in the callsgraph. It represents information about which of the arguments of the predicate can be called with domain variables, and which other predicates call P, respectively are called by P. The callsgraph will be described in more detail by an example in the following section 6.6.2.2.

6.6.2.2 The Static Analyzer

The task of this module is to complete the internal representation of the input program. Basically, that means to find out which predicates can be called with domain variables, especially which of their arguments can be called with domain variables of which domains. To know this is important because of the explicit representation of domain variables, which enforces FIDO to know where to expect them. The static analysis is done in two phases:

1. First, the information obtained by the explicit `define_domain` / 3 goals within the source program is used.
2. Starting from that information, the callsgraph is recursively traversed, propagating possible calls with domain variable arguments through it, until no further changes appear.

I would like to demonstrate the way static analysis works by the help of a small example. Look at the small program shown in figure 6.18: The callsgraph for this program after initialization, but before static program analysis, is shown in figure 6.19. After the first static analysis phase, using the information from the `define_domain` /3 goal, the callsgraph is modified as

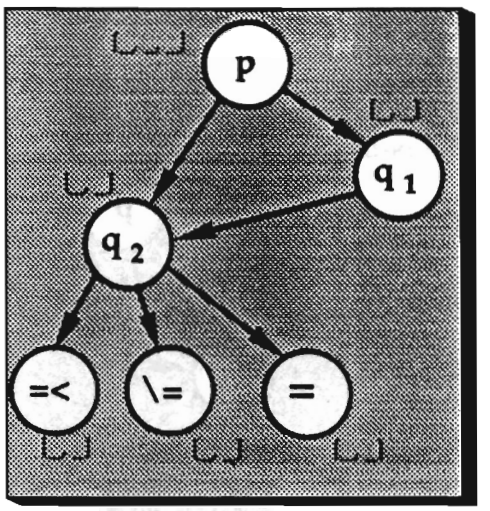

Figure 6.19: Initialized Callsgraph

shown in figure 6.20. In the second phase, starting from $p/3$, the domain variable appearances are recursively propagated through all descendants of p within the callsgraph. Figure 6.21 shows the state of the callsgraph after the propagation through the goal $q_1(X_1, Y)$ inside the definition of $p/3$. Figure 6.22 shows the final form of the callsgraph.

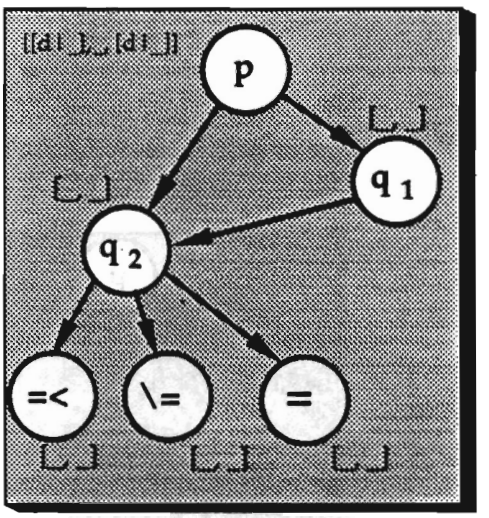

Figure 6.20: Intermediate Callsgraph (1)

The Callsgraph The information represented by the callsgraph can be used e.g. to detect that \= / 2 is not called with its first argument being a domain variable. Note however, two facts concerning the callsgraph:

1. The callsgraph entry for $\leq$ / 2 displays that this predicate can be called with both its arguments being domain variables. As a matter of fact, in the above example, $\leq$ / 2 will *never* be called with two domain variable arguments simultaneously. It will be called

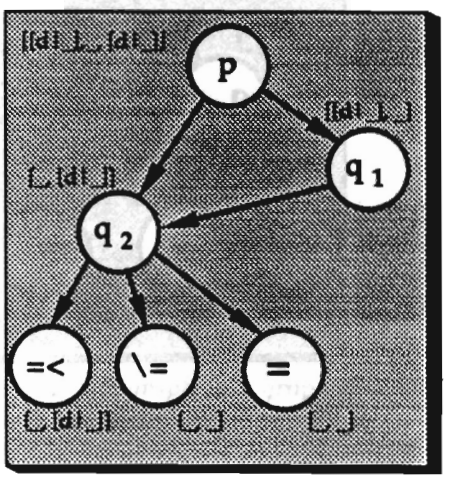

Figure 6.21: Intermediate Callsgraph (2)

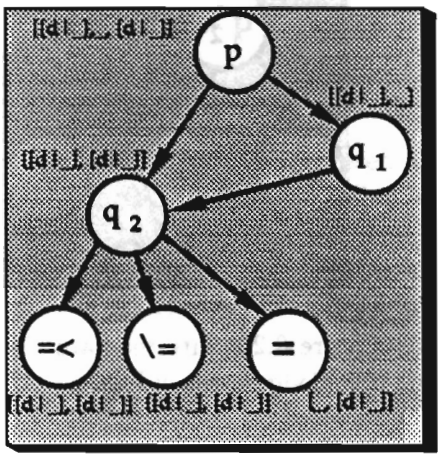

Figure 6.22: Final Callsgraph

either directly by $p/3$, in which case the second argument is a domain variable, or it will be called by q_1 /2. Then, the first argument is a domain variable, whereas the first one is not.

2. The nondeterministic predicate q_2 is represented by one single node in the callsgraph. The predicates called by q_2 are treated by the callsgraph as if they were conjunctive clauses inside a single goal.

The effect of these two simplifications is that the static analyzer will achieve overcautious results. It will predict predicate arguments to be called with domain variables which actually will not.
An alternative approach would be not to handle a predicate as a node, but to introduce a node for each alternative clause defining a predicate. That would make keeping track of each path through the program a much more difficult task. The additional knowledge gained by this refined strategy could be used in order to optimize code generation.
Nevertheless, for many examples, the simple analyzer implemented in FIDO-II does its work very well. Most important is that its analysis never yields wrong results. It will just do too much work, if domain variable definitions are used for a nondeterministic predicate. Especially, if some of the clauses for a predicate contain `define_domain` declaration while others do not, the domain variable arguments will be propagated through all goals appearing in the body of the definition, even if other clauses defining the predicate don't use domain variables, at all. That is because the callsgraph treats alternatives defining the same predicate as conjunctions. This is the effect shown in example 6.23, where the predicate > /2 is treated as if it was called with its first argument being a domain variable. Actually, since > /2 appears inside the second clause for $p/2$, it will never have a domain variable as its argument.

```
p(X, Y) :-
    define_domain(d, [X], [1, 2]),
    X \= Y.

p(X, Y) :-
    X > Y.
```

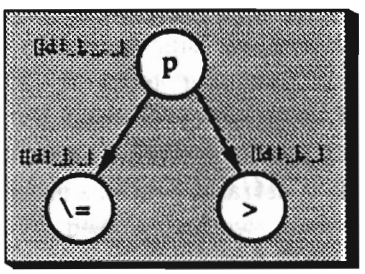

Figure 6.23: Overcautiousness of Callsgraph Representation

I don't think this is such a bad shortcoming of the system. It is hard to imagine that a nondeterministic definition of a predicate that uses domain variables in one case and that does not use them in another case makes much sense. At least, if the programmer is aware of the problem, it should be no problem to reformulate that by an explicit case distinction. For these

```
puzzle([K, N, O, W, H, D, F, I]) :-
    define_domain(digits, [K,N,O,W,H,D,F,I], 0..9),
    forward(1000*K + 100*N + 10*O + W + 100*H + 10*O + W =:=
            1000*D + 100*F + 10*K + I),
    instantiate([K, N, O, W, H, D, F, I]).
```

Figure 6.24: Another Formulation of KNOW + HOW = DFKI

regards, FIDO-II offers the built-in predicate **is_domvar** / 1 which succeeds if its argument is a FIDO domain variable.

6.6.2.3 The Constraint Normalizer

From the point of view of the system, the realization of constraint propagation by redefining constraints and formulating **delay** declarations on them turns out to be quite awkward for more complicated constraints. Therefore, a constraint normalizer is used in order to simplify constraints, i.e. to reduce them to a few basic formats that can be handled efficiently by predefined algorithms. In the current implementation, user-defined constraints are not submitted to normalization. It is the concern od the user to use them appropriately (see section 6.2.1.2). For built-in constraints, a normalizer is provided which allows the user a convenient formulation of constraints. This normalization is especially cut to arithmetic constraints. In section 6.2.1.1 I defined that arithmetic constraints must have a simple polynomic form. What the normalizer does, is converting *one* constraint C having a simple polynomic form into a set of "smaller" constraints C_i being of one of the patterns

- $C_i(X, Y)$,
- $C_i(X, Y + Z)$,
- $C_i(X, Y * Z)$ or
- $C_i(X, Y + N * Z)$,

where X, Y, Z, and N must be simple. Consider e.g. the kryptarithmetic puzzle $KNOW + HOW = DFKI$. A FIDO-II program solving that problem is shown in figure 7.3. That program reflects the arithmetic view, using carries in order to perform the arithmetic operations the way expected. For a programmer, however, it could be more convenient to formulate one big constraint instead of using one constraint for each adder operation. Figure 6.24 shows what that big constraint could look like. This is an example for a constraint in simple polynomic form. However, the constraint contains eight variables. Mentioning what I wrote before about constraint redefinitions, we would need 2^8 case distinctions only at level 2 of the redefinition process (see section 6.4.1.1), inducing an enormous requirement of memory. Thus, the constraint should better be transformed in a set of smaller constraints. The result of that redefinition is shown in figure 6.25.

The variables $T01$ to $T11$ are new variables used to connect the constraints.

6.6.2.4 The Code Generator

This module actually creates the destination program from the FIDO source code. It performs several tasks, using information from the callsgraph: It creates the domain and domain variable

```
{T01 =:=       K * 1000,
 T02 =:= T01 + N * 100,
 T03 =:= T02 + O * 10,
 T04 =:= T03 + W,
 T05 =:= T04 + H * 100,
 T06 =:= T05 + O * 10,
 T07 =:= T06 + W,
 T08 =:= T09 + I,
 T09 is  T10 + K * 10
 T10 is  T11 + F * 100,
 T11 =:=       D * 1000}
```

Figure 6.25: Constraint Set Resulting from Normalization of KNOW+HOW=DFKI

definitions necessary according to the `define_domain / 3` declarations in the source program. Each `define_domain` call will be replaced by calls to FIDO-II library predicates

`create_domain / 3` and `create_domvars / 3`,

creating the actual domains resp. domain variables. How these predicates are actually called, can be derived from the respective `define_domain / 3` goals. Assume there is a

`define_domain(DomId, Varspec, Domspec)`

declaration. It will be replaced by a `create_domain(DomId, DomProc,Dom)` and a `create_domvars(DomId, VarProc,Varlist)` call as follows:

- If `Domspec` is a list of domain elements, `Domproc` contains that list.
- If `Domspec` is a call to a predicate, `Domproc` contains this predicate call with `Dom` as its first argument.
- If `Varspec` is a list of variables, `Varlist` contains that list and `Varproc` is set to true.
- If `Varspec` is a predicate call of the form `genvar(`N`, `L`)`, `Varproc` is unified with this call and `Varlist` is unified with the resulting list L of domain variables.

For a better understanding of that, you should recall what I said about domain declarations in chapter 5.2
The second task of the code generator is to create the `?- compile(...)` declarations for the constraint redefinitions, or, in case of user-defined constraints, generate the redefinitions themselves. Finally, some built-in predicates in the source program have to be redefined if they are called with domain variable arguments. Otherwise, they could show an unexpected behaviour. Here again, the problem of variable duality appears as a general problem of FIDO-II. The explicit distinction between domain variables and non-domain variables causes problems, since it is a merely syntactical one (which stems from the internal representation), but in principle not a semantical one. I would like to show that by an example: assume e.g. that the source program contains a goal `write(`X`)` where X is a domain variable. The user is interested in the value of X, which is represented by Val_X, but not in the structure of the domain variable X. If we left the call as it is, `write / 1` would print the sixtuple representing the domain variable X as an answer substitution. There are two possibilities of handling that situation:

1. The programmer should be somehow aware of the way domain variables are represented by the system and avoid calling built-in predicates with domain variable arguments

2. The system can automatically recognize that case and replace the call to `write` /1 by a call to a redefined predicate

```
write_dv((&,_,_,_,ValX,_)) :-
    write(ValX).
```

Which solution should be preferred is a matter of opinion. Once more, there should be a trade-off between what must be burdened to the user and what can be done by the system. In the FIDO-II approach I chose the second way. Built-ins called with domain variables are automatically redefined the way shown above.
At the end of this chapter, the dynamic structure of the FIDO-II preprocessor is shown in figure 6.26. It incorporates the issues I mentioned concerning the single components. Note,

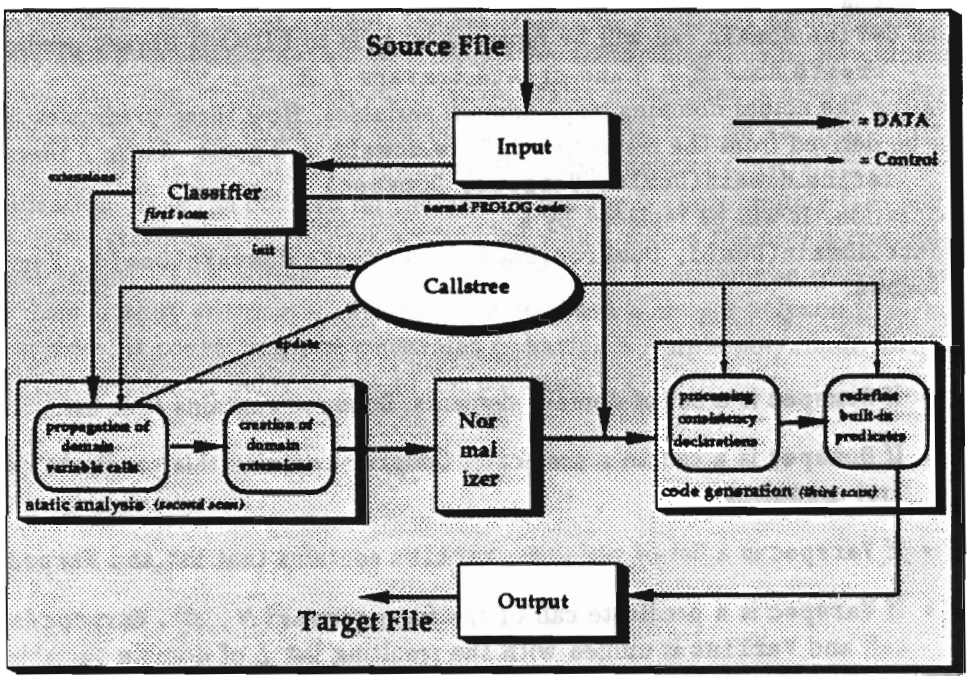

Figure 6.26: Dynamic Structure of the FIDO-II preprocessor

that three scans of the source program are made.

- The first one happens during reading the source, classifying it and initializing the calls-graph.
- The second scan (which is the most complex one and takes the greatest part of compilation time) is done for static analysis.
- By the third and last scan the source is modified due to the information obtained by the previous scans.

6.6.2.5 Conclusion

Up to here, I hope to have given an idea of how the preprocessor actually performs its tasks, transforming a source program containing additional information concerning domains and con-

sistency techniques into an efficiently executable SEPIA PROLOG program that can handle domain variables and achieves advanced control strategies according to the declarations in the source file.

6.7 Towards a FIDO-II Programming Methodology

In this chapter, I will try to give an idea of how FIDO-II programs should be written. The language provides facilities to integrate constraint solving techniques into logic programming. A central issue is, however, that the user has to make his/her contribution to that, namely by practising an adequate programming style. In FIDO-II, the constraint solver (or what we could call constraint solver there) is nested with the inference engine by the SEPIA **delay** mechanism. Thus, there is no clear inherent distinction between these two components in FIDO-II. That implies that the programmer has to care about handling constraints in an appropriate way. In the following, I would like to give some advice of how that can be done, providing an optimal use of the FIDO-II control mechanisms.

6.7.1 Doing Tests Before "Generates"

The idea of constraint solving is to propagate values through a constraint net, whose nodes are the variables and whose edges are the relations between the variables. Constraints shall be used as early as possible to support an early pruning of the search space. This implies the usefulness of the following general proceeding during constraint solving:

1. State the constraints as early as possible, i.e. first build the constraint net!
2. If not all variables can be instantiated by propagating value sets through the constraint net, start instantiating the variables as far as necessary.
3. In the following, use constraint propagation whenever possible.
4. If new constraints appear, add them incrementally to the constraint net.

Let's see now how we can integrate that methodology in FIDO-II. Obviously, point 1, which says to state the constraints first, can be fulfilled by writing the program constraints before instantiating the variables. Thus, FIDO programs should be written in a "test & generate" style instead of the usual "generate & test" manner. Since the selection of subgoals is done following the PROLOG strategy, i.e. top-down left-to-right, the programmer himself is responsible for stating the constraints in an appropriate order, calling the instantiation predicates thereafter. FIDO-II does not check whether the user does that the right way. Instead, if he decides to instantiate the variables before generating the constraint net, the resulting program will show a standard generate & test behaviour. Since the constraint variables will be instantiated before the constraints themselves are called, the call to a constraint will

- either succeed or
- immediately fail, thus imply backtracking,

but not be delayed. Then, the desired effect of using lazy evaluation in order to construct the constraint net and to obtain advanced control strategies, cannot be achieved. Thus we postulate

Law 1 (First FIDO Law) *Always state constraints before instantiating the variables appearing in the constraints*

If attention is given to this, the points 2 to 4 of the general proceeding shown above are guaranteed by the use of the **delay** mechanism for program control.

Instantiating Variables in FIDO-II

FIDO-II offers two built-in predicates in order to give values to domain variables. These are

- **instantiate** /1 and
- **instantiate_dl** /1.

Both predicates receive a list $[X_1, X_2, \ldots, X_n]$ as input and succeed if to each variable X_i, $1 \leq i \leq n$, a value from its domain can be assigned. **instantiate_dl** /1 achieves a first-fail heuristics, which has been described in more detail in section 6.5.
Having stated the constraints first, the call to **instantiate** /1 with the list of domain variables as its argument does the choices necessary to solve the problem or to detect its inconsistency. Furthermore, if **instantiate** /1 succeeds, we can be sure that the solution it has found is globally consistent.

6.7.2 Achieving Global Consistency

Using the FIDO instantiation predicates, we can enforce global consistency, respectively the detection of global inconsistencies within the constraint net. The notion of global inconsistencies in FIDO-II first came along with the problems caused by equality in FIDO-II (see section 6.2.5). Actually, it is not restricted to that special kind of constraint, furthermore it is a general problem, if no choices are made during problem solving in order to yield a complete and explicit solution of the constraint problem. In order to avoid inconsistent intermediate states of the constraint net, we have to postulate

Law 2 (Second FIDO Law) *Don't leave the domain variables uninstantiated, and propagate the constraint net as far as possible.*

Then, we can be sure (because of the soundness and completeness of the FCIR) to obtain globally consistent variable assignments.

6.7.3 Formulate the Strongest Constraints First

Due to the PROLOG selection strategy, there is a sequential generation of the constraint net. It turns out to be a wise decision to formulate strong constraints before constraints regarded as weaker. Strong constraints are constraints that are expected to have a strong pruning effect. Which constraints are actually considered as strong, actually depends on the problem to be solved. But, there are are some heuristics that can be often used as a guideline for that, e.g.

- equality constraints are very strong constraints, especially they are stronger than inequality constraints or ordering constraints,

- ordering constraints are often stronger than inequality constraints,
- `lookahead` declared constraints can often be used earlier than `forward` declared ones.

These observations can be subsumed by

Law 3 (Third FIDO Law) *Write strong constraints before weak constraints.*

The fact that the order constraints are formulated plays a role at all, goes against the declarativity paradigm. It is caused by the sequential goal selection strategy of PROLOG and by the way the SEPIA delay-stack is organized[26].

Conclusion Writing FIDO programs in the way described in this paragraph gives good support to an efficient use of consistency techniques in logic programming. Examples of FIDO-II programs for diverse example applications can be found in chapter 7.

[26] As a conclusion, we could postulate the **Fourth FIDO law**: a) don't think the order you write down constraints has no effects on performance!, and b) don't think it's declarative!. But this is awful, so it's just a footnote.

Chapter 7

Applications

After the theoretical and practical framework has been introduced in the previous chapters, I would like to give a feel for how FIDO-II can be used to support the solution of discrete combinatorical problems. Computational results will be compared with results yielded by other CLP systems. A special intention of this chapter is to show what FIDO-II can do, and what lies beyond its scope. In the following, some applications are shown for solving problems which are instances from thes following problem classes:

- Logical puzzles, e.g. n-queens, five-houses, kryptharithmetic puzzles, crossword puzzles.
- Graph colouring problems.
- Scheduling problems.

The run-times for all FIDO programs were taken on a SUN/4 SPARC station. The CHIP run-time results stem from van Hentenrycks book [van89a] and were measured on a VAX-785.

7.1 Logical Puzzles

The problems presented in this chapter can be characterized as problems where, starting from a finite (but big) set of possible combinations, one or a few have to be found. I present some well-known puzzles, such as the n queens problem, the five houses problem, the SEND+MORE=MONEY puzzle or a crossword puzzle. The importance of these puzzles, which are of small to medium complexity, is founded by two issues:

1. Their use as benchmarks comparing the computation results of different systems.
2. As vanHentenryck points out [van89a], if a system can't solve these problems within an acceptable time, how can it ever be able to solve more complex, real-world problems?!

7.1.1 The n-queens Problem

7.1.1.1 The Problem and its Solution

The task of this well-known problem is to place n queens safely on an $n \times n$ chessboard. "Safely" means that no queen must be threatened by another one. The solution to that problem can

be represented as follows:
The task is to find a safe row for each queen $X_i, 1 \leq i \leq n$. That way, we can implicitly express the constraint that each queen must be placed into a column of its own.
The constraints of that problem have already been explained in chapter 6.2.3.1. Program 7.1

```
queens(N, L) :-
    define_domain(queens, gen_var(N, L), gen_int_dom(D, 1, N)),
    safe(L),            \* generate constraint net *\
    instantiate_dl(L).  \* instantiate variables *\

safe([]).
safe([X]).
safe([X, Y|Z]) :-
    no_attack(X, [Y|Z]),
    safe([Y|Z]).

no_attack(X, Y) :-
    no_attack(X, 1, Y).

no_attack(X, N, []).
no_attack(X, N, [H|T]) :-
    regular(X, N, H),
    N1 is N + 1,
    no_attack(X, N1, T).

regular(X, N, Y) :-
    forward(Y =\= X),
    forward(Y =\= X + N),
    forward(Y =\= X - N).
```

Figure 7.1: A FIDO-II Program for the n Queens Problem

shows how a solution to that problem can be formulated in FIDO-II, using **forward** declarations on the constraints. There are three things to be stressed in that program:

1. The **safe** /1 predicate which states the safeness constraints for the queens is placed **before** the **instantiate** /1 predicate. Thus, the *test & generate* paradigm described in chapter 6.7 can be implemented.

2. The **define_domain** goal defines a dynamic domain using a number of variables that will be determined at run-time (see chapter 5.2.2).

3. The **forward** declarations denote that their arguments shall be executed in a forward-checking manner (see chapter 6.2.3). I pointed out that forward-checking is well-suited for inequality constraints (chapter 6.2.3.5).

4. Variable instantiation is performed using a first-fail heuristics.

7.1.1.2 Computational Results

In the following, I will compare the run-time for finding the first solution of the n queens problem needed by FIDO-II and I will compare them with the times needed by other systems.

N	F21	F22	F23	F24	F25
4	0.03	0.05	0.03	0.07	0.02
8	1.10	1.12	0.48	0.53	0.14
12	4.61	4.10	1.2	1.43	0.35
16	50.4	2.03		0.83	0.57
32	??	12.20	??	3.52	3.08
48	??	59	??	10.2	8.55
96	??	?	??	122.8	66.5

Table 7.1: Results for Several FIDO-II Versions for N Queens

Solving the n queens problem efficiently is not only a matter of the system used, but it is also greatly determined by the representation of the problem within one system. Table 7.1 compares the following versions of FIDO-II programs for solving the n queens problem for some n:

1. F21: a FIDO-II forward-checking program using the normal instantiation predicate `instantiate` /1.
2. F22: the program shown in figure 7.1.
3. F23: the same program as F21, but for each n, an explicit domain definition of the form `1...n` is made. Thus information about the domain structure can be exploited.
4. F24: the same program as F23, but variables are instantiated using a first-fail heuristics (see chapter 6.5).
5. F25: the program is basically the same as F24, but besides the first-fail heuristics, we perform variable instantiation not from left to right, but from the center of the chessboard to the left and to the right. That is achieved by ordering the variable list which is input argument for the `instantiate_dl` /1 predicate the following way (e.g. for $n = 8$):

$$\texttt{instantiate\_dl}([X5, X4, X3, X6, X2, X7, X1, X8]).$$

Thus, we see that in FIDO-II, the efficiency of solving the n queens problem depends drastically on the variable instantiation order. Table 7.2 compares the best FIDO-II results to the results yielded by the following programs:

1. The generate & test program for the n queens problem from figure 1.1.
2. The standard backtracking program from figure 1.2.
3. A FIDO-I program for solving the n queens problem.
4. A CHIP program run on the CHIP prototype interpreter. That is denoted as "CHIP(1)" in the table.
5. Recent computational results for CHIP, denoted as "CHIP(2)".

The recent CHIP run-time results were taken from [Tv89]and run on KCM. A "?" in the tables means that the run-time is not known to the author. A "??", however, means that computation takes too long (i.e. more than ca. 1000 cpu seconds).

N	G&T	S.B.	FIDO-I	FIDO-II	CHIP(1)	CHIP(2)
4	0.1	0.08	0.9	0.02	0.09	?
8	?	0.73	6.0	0.14	0.77	?
12	?	3.75	?	0.35	1.74	?
16	?	620	??	0.57	1.09	0.01
32	??	??	??	3.08	4.05	?
48	??	??	??	8.55	?	?
64	??	??	??	17.42	14.05	?
96	??	??	??	66.5	36.23	?

Table 7.2: Results for the N Queens Problem

7.1.1.3 Evaluation of the Results

The tables 7.1 and 7.2 allow the following conclusions:

1. FIDO-II yields the expected improvement compared with FIDO-II.
2. Using domain-specific information in FIDO brings drastically better results (compare programs F22 and F24!).
3. A good instantiation order, which is achieved by using a first-fail heuristics (or the heuristics to start from the mid of the chessboard) has astonishing positive effects on the performance of FIDO-II.
4. FIDO-II outperforms standard PROLOG strategies (such as G&T, Standard Backtracking) already for relatively small problems.
5. For small- or medium-sized applications, FIDO-II can compete with early CHIP versions.
6. For more complex problems, the deeper integration of domains and consistency techniques in CHIP pays off.
7. Recent CHIP clearly outperforms FIDO-II.

The greatest n for which the n queens problem could be solved by FIDO-II, has been 96. For the 128 queens problem, even after three hours, no solution has been computed. The limitations of FIDO-II become quite clear once you compare that to the recent CHIP results. As is pointed out in recent papers, CHIP can actually solve the 1500 queens problem.

7.1.2 The Five Houses Puzzle

7.1.2.1 The Problem and its Solution

The Five Houses Problem can be described as follows: Five people with five nationalities live in the first five houses of a street. Each of them has a different profession, animal and has a favourite drink. Besides, the five houses are painted differently. It is known that the englishman lives in a red house, the spaniard owns a dog, the japanese is a painter, the italian drinks tea, the owner of the green house drinks coffee, the sculptor breeds snails, the diplomat

lives in the yellow house, the owner of the middle house drinks milk, the violinist drinks fruit juice, the norwegian lives in the first house on the left and the green house is on the right of the white one. The problem is made more complicated by the following disjunctive constraints: the norwegian lives next to the blue house, the fox is in the house next to the doctor's and the horse is in the house next to that of the diplomat.
The question is, now, who owns a zebra and who drinks water[1]. A FIDO-II program for solving that puzzle using forward-checking is shown in figure 7.2. The following issues should be noted

```
houses( [England, Spain,   Japan,  Italy,   Norway,
           Green,   Red,  Yellow,   Blue,    White,
         Painter, Violinist, Diplomat, Doctor, Sculptor,
            Dog, Zebra,     Fox, Snails,    Horse,
          Juice, Water,     Tea, Coffee,     Milk]) :-
    define_domain(houses, [England, Spain, Japan, Italy, Norway,
                           Green, Red, Yellow, Blue, White,
                           Painter, Diplomat, Violinist, Doctor, Sculptor,
                           Dog, Zebra, Fox, Snails, Horse,
                           Juice, Water, Tea, Coffee, Milk],
                           1..5),
    Norway = 1, Milk = 3,
    neighbour(Norway, Blue),
    neighbour(Fox, Doctor),
    neighbour(Horse, Diplomat),
    forward(Green =:= White + 1),
    England = Red, Spain = Dog,
    Japan = Painter, Italy = Tea,
    Green = Coffee, Sculptor = Snails,
    Diplomat = Yellow, Violinist = Juice,
    all_different([England, Italy, Spain, Norway, Japan]),
    all_different([Green, Red, Yellow, Blue, White]),
    all_different([Painter, Diplomat, Violinist, Doctor, Sculptor]),
    all_different([Dog, Zebra, Fox, Snails, Horse]),
    all_different([Juice, Water, Tea, Coffee, Milk]),
    instantiate([England, Spain,   Japan,  Italy,   Norway,
                 Green,   Red,  Yellow,   Blue, White,
                 Painter, Violinist, Diplomat, Doctor, Sculptor,
                 Dog, Zebra, Fox,    Snails, Horse,
                 Juice, Water,     Tea, Coffee,   Milk]).

neighbour(X, Y) :-
    forward(X =:= Y - 1).
neighbour(X, Y) :-
    forward(X =:= Y + 1).

all_different([]).
all_different([X|Y]) :-
    out_of(X, Y),
    all_different(Y).

out_of(_, []).
out_of(X, [Y|Z]) :-
    forward(X =\= Y),
    out_of(X, Z).
```

Figure 7.2: A FIDO-II program Solving the Five Houses Problem

about the problem and its solution::

[1]There exists a unary solution for that problem: the spaniard owns the zebra, and he also drinks water.

Algorithm	Min. Runtime	Avg. Runtime	Max. Runtime
SB	1.28	82.06	717.1
F1	?	14.6	?
F21	0.22	0.70	2.16
F22	0.30	0.65	1.13
CHIP	0.24	1.49	4.95

Table 7.3: Results for the Five Houses Problem

- The problem is represented by numbering the houses from left to right, thus, yielding a very simple domain which ranges from 1 to 5.
- Here, we use the normal instantiation predicate **instantiate** /1, since it is faster than using first-fail heuristics. That shows that first-fail is not appropriate to all cases!
- Although the variables used in the equations are domain variables, the general unification operator = /2 can be used. The FIDO-II preprocessor recognizes and maintains the appearance of domain variables. In the five houses example, however, since there are only domain variables from a single domain, the restricted domain variable unification operator = /2 could be used. I described that operator in chapter 6.2.5.2.
- The order the constraints are written embodies the heuristics that strong constraints should be formulated before weak constraints (see chapter 6.7).

7.1.2.2 Computational Results

We compare the following program concerning their run-time performance:

1. SB: a standard backtracking program.
2. F1: a FIDO-I program for solving the five houses puzzle.
3. F21: the FIDO-II program shown in figure 7.2.
4. F22: the same program as F21, but using a first-fail heuristics.
5. CHIP: A forward-checking CHIP program solved by the CHIP prototype interpreter.

The run-time results of the programs are shown in table 7.3. Since the run-time needed for solving the problem largely depends on the variable instantiation order, I implemented an instantiation predicate producing random instantiation orders and took the average of about 100 instantiation orders in order to get expressive results. Looking at table 7.3, we can summarize the following interesting notions:

1. FIDO-II clearly outperforms FIDO-I and the standard backtracking program.
2. For the five houses problem, the use of first-fail heuristics doesn't pay off the way it did e.g. for n queens. Both algorithms yield almost the same results. First-fail soothes the negative effects of awkward instantiation orders a bit, but doesn't pay off as regards the average runtime.

3. FIDO-II yields better results than the CHIP prototype interpreter. However, that must not be assessed too positive for FIDO-II. First, the five houses problem is a relatively simple problem, and we saw, that for simple and medium sized problems, FIDO-II is well-suited. Second, recent CHIP versions will show a greatly improved performance, and, third, the machines the programs run on are different (see chapter 7.1.1).

7.1.3 Cryptharithmetic

7.1.3.1 The Problem and its Solution

A well-known problem is the SEND+MORE=MONEY puzzle. The letters shall be replaced by digits so that the addition

```
    S E N D
+   M O R E
  ----------
= M O N E Y
```

gives a correct result. The problem can be formulated by defining an adder for each column, returning a carry and a value. Figure 7.3 shows a FIDO-II program achieving forward-checking control for a slight variation of that example, for the

```
   K N O W
+    H O W
----------
=  D F K I
```

puzzle. The following interesting issues about that program should be mentioned:

```
puzzle([K, N, O, W, H, D, F, I],[C1,C2,C3]) :-
    define_domain(digits, [K, N, O, W, H, D, F, I], 0..9),
    define_domain(carry, [C1,C2,C3], 0..1),
    forward(K =\= 0),
    forward(H =\= 0),
    forward(D =\= 0),
    all_different([K, N, O, W, H, D, F, I]),      \* see five houses problem *\
    forward(C1 + K     =:= D),                    \*  adder conditions *\
    forward(C2 + N + H =:= F + 10 * C1),          \*    ----  "  ----  *\
    forward(C3 + O + O =:= K + 10 * C2),          \*    ----  "  ----  *\
    forward(     W + W =:= I + 10 * C3),          \*    ----  "  ----  *\
    instantiate([C1,C2,C3, K, N, O, W, H, D, F, I]),
```

Figure 7.3: A FIDO-II Program for KNOW + HOW = DFKI

1. The problem is solved by forward-checking use of the adder constraints. In [van89a], a looking-ahead use of constraints is proposed. Since FIDO-II does not provide looking-ahead for inequality and =:= /2, we can't do that here.

2. Variable instantiation is performed using a first-fail heuristics.

Algorithm	Min. Runtime	Avg. Runtime	Max. Runtime
SB	10.25	2874	25140
F1	?	22.4	?
F21	0.25	?	633.1
F22	0.16	1.18	5.4
CHIP	?	0.08	?

Table 7.4: Results for the SEND+MORE=MONEY Puzzle

3. The two domains **digits** and **carry** are defined by two seperate define_domain calls.

4. The inequality constraints are formulated before the equality constraints. At the first look, that seems to be an exception from the third FIDO law (see chapter 6.7), but, since one argument of the = \ = /2 constraints is ground from the beginning, it can be considered as stronger than the equality constraints, which contain more unbound variables.

7.1.3.2 Computational Results

Table 7.4 shows the run-times for the SEND-MORE-MONEY puzzle for the following algorithms:

1. SB: A standard backtracking algorithm[2].

2. F1: A FIDO-I program using forward-checking.

3. F21: A FIDO-II program using forward-checking and normal instantiation ordering.

4. F22: Like F21, but using a first-fail heuristics.

5. CHIP: A looking-ahead program on the CHIP prototype interpreter.

The results allow the following conclusions:

1. FIDO-II outperforms FIDO-I.

2. The use of first-fail heuristics in FIDO-II is adequate for this problem.

3. CHIP yields much better results, since looking-ahead is a more appropriate strategy for cryptarithmetic problems than forward-checking is. There is no information available about the time a CHIP forward-checking program would need.

4. Instantiation ordering is very important in that example. A forward-checking program without first-fail heuristics will be not much better than standard backtracking, if an awkward instantiation order is chosen.

The KNOW+HOW=DFKI problem has been solved in 0.87 seconds by the FIDO-II program shown in figure 7.3.

[2]The average execution time on 50 instantiation orderings was taken.

```
\* substring(Word, Position, Length, Subword) succeeds if Subword is the  *\
\* substring of Word which begins with the 'Position'th character and has *\
\* length Length. (SEPIA built-in predicate)                               *\
sameletter(W1, I, W2, J) :-
    substring(W1, I, 1, Letter),
    substring(W2, J, 1, Letter).
```

Figure 7.4: The **sameletter** /4 User-Defined Constraint

7.1.4 Crossword Puzzles

7.1.4.1 The Problem and its Solution

Crossword puzzles can be represented as constraint problems, since the words in the puzzle's rows and columns are constrained to have letters in common. That relation can be expressed by a constraint **sameletter(Word1, I, Word2, J)**, which succeeds *iff* the Ith letter of the word Word1 is equal to the Jth letter of the word Word2. The definition of the **sameletter** /4 constraint in figure 7.4. I implemented the crossword puzzle example used in [van89a], p. 138f, and provided a foward-checking use of the **sameletter** /4 constraint. That example contains 146 words and induces 1902 constraints.

7.1.4.2 Computational Results

In FIDO-II using a first-fail heuristics, the problem has been solved in about 16.4 seconds. The same program using normal variable instantiation needs 14.4 seconds in order to solve the problem[3].
Besides, a precompile time of ca. 70 seconds must be added. The CHIP prototype interpreter needs 48 seconds to find a solution of the problem, using forward-checking, and a little more than one minute, using looking-ahead. Note that the constraint **sameletter** /4 is an example of a user-defined constraint as described in chapter 6.2. The redefinition of that constraint is created automatically during precompilation. Therefore, a lot of redefinition code is generated, exploding the size of the output file to about 330 kilobytes. Besides, the big number of domain variables (146) makes preprocessing going quite slowly, as the precompilation time reveals. That clearly shows the limitations of FIDO-II facing bigger-sized problems. The listing of the FIDO-II program solving the crossword-puzzle can be found in appendix A.1.

7.2 Graph Colouring

Many important real-life problems can be represented as instances of graph colouring problems. Examples of such problems are operations research applications as warehouse location problems or production scheduling. One instance of graph colouring is colouring a geographical map:
given a map which shows several regions (countries), the task of graph colouring is to colour each region with one out of a fixed number of colours. The important constraint is that no two adjacent regions must have the same colour.
A FIDO-II program which colours the map of Europe is shown in appendix A.2. The program

[3] However, the program using first-fail heuristics is certainly more stable for bad instantiation orders.

contains 74 inequality constraints and can be solved very efficiently by a forward-checking algorithm. Moreover, the efficiency of the FIDO-II algorithm for that problem can be essentially influenced by an appropriate representation. That representation should be tuned w.r.t. the FIDO-II implementation peculiarities: we've seen that some domain types, namely integer subsets $\{n, n+1, \ldots, m\}$, defined as $n \ldots m$ in the **define_domain** call, can be processed very efficiently. Assume we want to use four colours, which corresponds to a set $\{yellow, red, green, blue\}$. By representing each colour by an integer, using the set $\{1, 2, 3, 4\}$ instead, efficiency can be remarkably increased. The following run-time results were gained:

- Using normal instantiation and an integer subset representation of the domain, the map was coloured by FIDO-II in 0.06 seconds.
- Using a first-fail heuristics, the task could be solved in 0.21 seconds. Here, we have another example where first-fail heuristics does not pay off[4].
- Using explicit domain representation (i.e. the domain has been represented as {**yellow**, **red**, **green**, **blue**}), FIDO-II needed 0.09 seconds respectively 0.25 seconds with first fail.
- Using the standard instantiation predicate, it took the program about 0.9 seconds to find out that the map could not be coloured by using only three different colours. Using a first-fail heuristics, the run-time was 1.5 seconds.
- The time needed for precompiling the program has been about 4 seconds.

. The only comparison available stems from [Hol90b]. A program written in Holzbaur's META-Prolog, which is based on meta-interpretation, has coloured the map within 1.75 seconds[5] on an APOLLO workstation. As I pointed out in chapter 6.2.3.5, inequality constraints are very well-suited for forward-checking use.

7.3 Scheduling Problems

Scheduling problems are an important real-world application for discrete combinatorical problems. In this section, an idea shall be presented how to solve the following handcrafted scheduling problem in FIDO-II.

7.3.1 The Problem

In the computer department of an enterprise nine people, *Harry*, *Sally*, *Fred*, *Isabel*, *Paul*, *Pamela*, *Tina*, *Anna*, and *Mary* are employed. Since we are in the nineteen-nineties, the staff has to work only three days a week. In order to strengthen the motivation of the employees, the executive has decided to introduce a three-shift system. On each shift, exactly three employees have to be present. Now, the task is to manage the schedule for these three days, satisfying the following constraints:

1. Nobody may work during two subsequent shifts.
2. On each shift, exactly three persons must be present

[4] The main reason for that is the small domain-size of 4 in this problem.
[5] at a time when the map of Europe was still much easier to colour!

3. Each employee must do exactly three shifts a week.

These are the constraints given from the enterprise management, which turn the problem hard enough. You, however, as a connaisseur of the department internals are to know some additional constraints that keep your uneasy feeling increasing:

4. Harry and Sally are married. That's why they would like to have the same working schedule.
5. Fred and Isabel are married. That's why they would *not* like to have shared shifts.
6. Pamela is afraid of Paul. That is why she does neither want to have her shifts directly before Paul nor together with Paul or directly after Paul.
7. Tina hates Mary and therefore, you'd better not give these two for the same shift.
8. Since Paul seems unable to remember his password and Anna is the only person to know it, you should always give her the shift before Paul, so she can tell him the password.
9. Harry is always frightened during the night. Because you are a wise man, you decide never to give him the last shift of the day.
10. Since Tina has two little children, but no husband, she'd like to be at home n in the morning. Thus, she should not be given the first shift.
11. Mary does likes to work neither in the morning nor during the night. Therefore, it should be tried to give her the second shift.
12. Since Fred is in love with Pamela, his creativity will greatly benefit from working together with her.

This is what the situation looks like. Now, you should try to find an appropriate schedule satisfying all the constraint, the official ones and the inofficial ones.

7.3.2 The Solution to the Problem

There are many ways to represent the problem as a constraint problem over finite domains. The main problem with it is that some constraints refer to persons (e.g. constraint no. 4 to 11.), while other constraints refer to shifts, such as constraint no. 2.

- For constraints referencing persons, it would be better to define for each person one domain variable, ranging over the number of shifts.
- For shift constraints, it would be favorable to have a domain variable per shift, which ranges over all triples of workers which can be put on that shift.

I chose the first representation, since there are more person constraints than shift constraints in the examples and since the domains are smaller that way. I tried to represent the shift constraints in an implicit manner, e.g. constraint 3 by introducing five variables per person, implying the five shifts. However, t find a good possibility to represent the constraint number 2, which gives expression to the fact that on each shifts, there must be exactly three

members. The only possibility I found was to express this constraint as a test after all active constraints have been executed. For this, I implemented a special instantiation predicate `instantiate_card(List, Dom, Cardinality)`, which instantiates a list *List* of domain-variables with values from [1...*Dom*], testing that each value must not be assigned to more than *Cardinality* variables. Note that this merely implements a standard backtracking search on the pre-pruned search space.The prepruning has been done by the other constraints, beforehand. This is much better than solving the whole problem with standard backtracking search, but is not convincing for more complex problems. The above program is much too slow for computing a four-day schedule or even a five-day schedule, since too much backtracking has to be performed[6].
In order to remedy this, it would be very useful to implement a forward-checking use of a cardinality constraint as it is proposed in van Hentenrycks CC(FD) (see chapter 2) for these issues. Thus, by cardinality restrictions, values could be actively removed from the domains. This, however, is not yet performed in FIDO-II. The complete FIDO-II program finding a solution to the scheduling problem can be found in appendix A.3.

7.3.3 Computational Results

FIDO-II has found a solution to the problem in 48.3 seconds using the specialized instantiation predicate `instantiate_card` /3 described above. Since the example has been 'hand-made', there is no comparison possible to other systems. FIDO-II precompilation time for the scheduling example was 2.8 seconds.
I already mentioned the problems connected with this example: the fact that a standard backtracking component is introduced again makes the program too inefficient for handling more complex scheduling problems. As a conclusion we can formulate that these problems lie beyond the capacities of FIDO-II, since they imply too many constraints and too many variables, even if smaller problems, such as the example above, can be handled satisfactorily. However, a possibility to master a bigger class of scheduling problems would be to integrate the active a priori use of cardinality constraints into FIDO.

7.4 Conclusion

In this chapter, the applicability of finite domain constraints in FIDO-II has been shown by some examples belonging to different problem classes. We've seen that, for small- and medium-sized problems, FIDO-II yields good results which are comparable to the performance of early CHIP versions. For very complex real-world problems, however, which can be mastered by systems as CHIP, the performance of FIDO-II is not sufficient. The main reason for that fact is the expensive way FIDO-II handles domain variables (see chapter 5.2.1). Domain and consistency technique management are performed on PROLOG level, which makes them quite costly. Also the active use of cardinality constraints is not provided in FIDO.
As I stated several times before in this work, an approach providing deeper integration seems much more promising for handling really complex problems.

[6] Much too slow means that the program runs longer than one hour!

Chapter 8

Summary and Outlook

In this chapter, the main results of this work shall be summarized. I will briefly discuss the most important problems which occurred in connection with my work. This will show the scope of application, but also the main limitations of FIDO-II as an approach for integrating finite domain constraints in logic programming. Finally, an outlook will be provided concerning possible extensions of FIDO-II, and also concerning the further development of the FIDO project.

8.1 What Has Been Done?

In this work, I've been presenting the design and the implementation of the FIDO-II finite domain CLP system. FIDO-II provides a way of conveniently formulating discrete combinatorical problems using constraints. It allows variables to be defined as domain variables and predicates to be declared submitted to a forward-checking or looking-ahead execution strategy.The approach has started from an advanced-feature PROLOG system offering a coroutining mechanism. An explicit domain concept has been implemented in PROLOG and the advanced control strategies have been simulated by doing partial (lazy) evaluation using the coroutining mechanism.
The main differences between this work and other approaches towards constraint logic programming over finite domains have been outlined in section 2.2.6.

The approach turned out to yield tenable run-time results for small or medium sized problems (see chapter 7 for run-time results). But for real-life problems, limitations of the approach become evident. There are some main issues responsible for this, some are due to special representation and implementation issues, others due to more general reasons. These will be discussed in the following section.

8.2 What Remains to be Done?

In the following, some drawbacks of FIDO-II will be presented.

8.2.1 Explicit Maintenance of Domains and Domain Variables

The problems caused by the necessity of maintaining the domains on a layer on top of the PROLOG system are responsible for the difficulties described in chapter 6.2.5. The main

shortcomings are

- the expensive tests and case distinctions needed to localize and handle domain variable appearances, and
- the currently unsolved problem of how implicit unification of domain variables can be mastered.

8.2.2 Domain Representation

The realization of domains as uninterpreted functors with validity flags (see chapter 5.2) has some advantages, e.g. easy backtrackability. One disadvantage consists in the fact that operations on domains become high level PROLOG operations and thus, they become quite expensive. It would be preferable to represent domains as bit vectors, allowing fast (possibly machine-level) bit manipulation operations, such as bitwise Xor or Nand, to work on them. Experience with bit vector representation of domains in SEPIA has been made with FIDO-I. It showed that SEPIA is not well-suited for such representation, since the biggest number of bits that can be processed is 32 (maxint = $2^{32} - 1$). In order to represent bigger domains, lists of integers have to be maintained, so that much of the efficiency originally gained by making use of bit vectors is lost.
Shifting domain handling to machine (or at least pseudo-machine (WAM)) level could bring an immense improvement in efficiency. The second aspect concerning domains in FIDO-II is that semantically equivalent domains are not always treated equally by the system if they are enumerated in different orders. e.g. the domains $[1, 2, 3]$ and $[3, 2, 1]$ have an internal representation in FIDO-II which is not unifiable. This problem could have been solved by pre-sorting the domains, but it has not been provided in FIDO yet. In general, it is by no means a trivial problem to check whether two domains are equal[1].

8.2.3 Dual Variable Concept

A conceptual drawback of FIDO-II is the distinction made between domain variables and "normal" variables. This duality is awkward for semantical reasons, since even normal variables *are* domain variables in fact, but not over finite domains. This semantical aspect is not sufficiently integrated into FIDO-II. In CHIP e.g., a flag "h" is provided for denoting Herbrand logic variables. Thus, a duality is introduced which should not be subject to the user's view.

8.2.4 Declarative Semantics

An important aspect of logic languages is their dual semantics. Thus, for a language as FIDO-II, which is based on PROLOG , the question of an exact declarative and procedural semantics should be examined. The procedural semantics of FIDO-II, which is essentially covered by the constructs `define_domain` /3, `forward` /1 and `lookahead` /1, is obvious. It is defined by the preprocessor that transforms appearances of these goals (which are rather declarations, from a logical point of view) into new PROLOG code in an algorithmically specified manner. The declarative semantics of these structs, however, requires a closer look. The first awkward fact is that the FIDO extensions are formulated as PROLOG goals (mainly for reasons of convenience of implementation), but a FIDO program cannot be executed by PROLOG because

[1] e.g. is the domain $[1, 2, 3, 4]$ with the elements 3 and 4 removed equal to the domain $[1, 2]$?

define_domain /3, **forward** /1, **lookahead** /1 are no defined PROLOG predicates. They rather give information to the preprocessor about how to handle control over the program. Moreover, they localize where domain variables are used, and are replaced by PROLOG goals later on by the preprocessor. Thus, from the viewpoint of PROLOG declarative semantics, the effects of these language structs could be best described as side-effects, i.e. they always succeed and fulfill their main tasks through side-effects. But I am not too happy with the word *side-effects*, because FIDO extensions are not PROLOG predicates, thus they are not evaluated by PROLOG, and therefore cannot have side effects in a PROLOG sense. They simply provide preprocessing information.
Formulating the FIDO-II extensions as real PROLOG predicates which use output variables would have been nothing more than syntactic sugar. However, it would have facilitated solving the question of a declarative semantics of the FIDO extensions. In this work, not too much thought has been given to this issue. Since the transformations done by the preprocessor have a clear procedural semantics, whose soundness and completeness have been shown (domain concept, forward-checking), I am not too concerned about this point. However, finding a clear declarative semantics for FIDO (which could probably be made easier by some changes in the language) seems to be subject of future work.

8.2.5 Constraint Types

The distinction between built-in constraints and user-defined constraints (se chapter 6.2) is also an arbitrary one, since it only exists from the viewpoint of the system. The remarkable differences in efficiency caused by using or not using built-in constraints can be regarded as a shortcoming of the current implementation and could be reduced to an acceptable limit by using domain specific information for non-built-in constraints, too, and by increasing the scope of built-ins. For the time being, the user must be aware of the two types and their particuliarities and use built-in constraints whenever it is possible.

8.2.6 Nesting of Inference Engine and Constraint Solver

In FIDO-II, the constraint solver and the advanced control strategies achieved by linking the inference engine to the solver are only simulated with the help of the coroutining mechanism. There is not such a clear physical modularization (of FIDO-II code) into an inference part and a constraint solving part as in other CLP systems (see figure 2.1). In FIDO, both issues are left to the PROLOG system and can be influenced merely in an indirect manner, i.e. by formulating **delay** declarations on constraints. The distinction between inference engine and constraint solver can only be made on a logical level. This makes the way control is brought about in FIDO-II a little difficult to understand. Yet, it is a general shortcoming caused by the approach itself, not by its implementation.

8.2.7 Weak Constraints and Relaxation

FIDO-II is essentially based on first-order logic. Therefore, all constraints that appear in FIDO-II programs are **hard** constraints, i.e. all constraints appearing in the body of a goal definition have to be satisfied in order to make the goal succeed. In many real-life applications, however, there are no such "black & white" problem solving strategies. Assume, e.g. we want to use a constraint system for the tool selection module of a program that transfers a CAD drawing of a lathe part into a CNC program. The latter shall manufacture automatically the

lathe part specified by the drawing[2]. Implementing a good tool selection program based on hard constraints only is impossible for several reasons, e.g.

- It is surely desirable to manufacture as many features of a lathe part as possible with the same tool, but this is certainly not a hard constraint.
- It is often helpful to weigh alternatives in order to express preferences, e.g. "if tool A is available, we will use it, if not, we will use tool B instead", or "for roughing, we will better use a tool with a big edge angle, but if we can avoid a change of tools by using one with a smaller one, we will do that!".
- If no tool is found that satisfies the problem constraints, some of them will have to be relaxated in order to find a solution.

In FIDO-II, it is hardly possible to achieve these more flexible control strategies since the maintenance of constraints is the task of the PROLOG system, which can hardly be influenced by FIDO without loosing much of the efficiency that was gained by leaving the constraint handling to the system. However, it will be subject to further research.

8.3 Outlook

As we have seen in the last paragraph, the main shortcoming of FIDO-II is caused by the necessity of explicitly representing and maintaining domain variables at the PROLOG level. It seems that a deeper integration of these features could yield considerable improvements. Basically, I see two approaches going in this direction:

- Automatizing unification using a mechanism like the meta-terms presented in chapter 5.3.
- Providing a deeper integration of domains (and of consistency techniques, too) into the WAM.

The main problem with the first point is how it can be generalized w. r. t. using meta-terms for handling arbitrary constraints, thus allowing an efficient processing of the domains.
The second approach is pursued in the third part of FIDO, FIDO-III (see chapter 4.2). There have not been any results available concerning FIDO-III yet. Thus, my expectations as regards a better run-time behaviour are purely speculative. But, looking at the performance of systems based on similar principles (e.g. CHIP), where changes were made in the underlying abstract machine, the approach seems promising.
After we will have implemented FIDO-III, and after some experience with it will have been gained, we will be able to study further refinements and extensions of it. For example, the integration of weak constraints and of relaxation techniques will be much simpler in a deeply integrated system such as FIDO-III, since there, constraints can be handled explicitly.
A further prospect is the integration of hierarchical domains in FIDO. Thus, structural information can be used to improve system performance. An idea in this context would be e.g. a coupling with the KL-ONE-like taxonomic language TAXON [ADH91], which has been developed in the ARC-TEC project at DFKI. These will be the tasks for the future.

[2]such a "CAD2NC" project is currently under development at the ARC-TEC project at DFKI, Kaiserslautern, Germany.

Appendix A

FIDO-II Example Programs

A.1 The Crossword Puzzle

In chapter 7.1.4, it has been shown how FIDO-II can be used to solve crossword puzzles. In the following, you find the source code of the crossword puzzle program from [van89a].

```
cwp([W10, W28, W70, W92, W131],
    [W12, W15, W30, W32, W33, W38, W41, W45, W49, W55, W57, W60, W76, W77, W81, W85, W94, W95,
     W101, W108, W122, W126, W127, W129, W134, W141, W142],
    [W18, W22, W25, W31, W34, W39, W40, W43, W53, W56, W62, W63, W64, W68, W71, W73, W75, W83,
     W102, W105, W106, W111, W113, W116, W117, W121, W128, W132, W137, W138, W140, W145, W146],
    [W3, W4, W5, W9, W16, W21, W23, W26, W27, W44, W47, W52, W65, W66, W69, W72, W79, W86,
     W87, W91, W93, W98, W99, W100, W107, W109, W125, W130, W133, W139],
    [W7, W8, W11, W14, W29, W48, W67, W74, W82, W89, W103, W120, W124, W136, W143],
    [W2, W37, W50, W54, W59, W61, W80, W84, W8, W90, W96, W97, W112, W114, W118, W119, W123],
    [W1, W6, W20, W35, W36, W42, W51, W110, W115, W135, W144],
    [W13, W17, W19, W104],
    [W24, W46, W58, W78]) :-

    define_domain(word2, [W10, W28, W70, W92, W131], ["ci", "id", "in", "ir", "le"]),
    define_domain(word3, [W12, W15, W30, W32, W33, W38, W41, W45, W49, W55, W57,
                          W60, W76, W77, W81, W85, W94, W95, W101, W108, W122,
                          W126, W127, W129, W134, W141, W142],
                         ["ain", "ave", "bis", "bol", "cou", "eau", "eta", "feu",
                          "gre", "ils", "ios", "les", "lie", "lue", "mer", "nep",
                          "nie", "ohe", "ole", "olt", "ost", "oui", "sem", "sic",
                          "tue", "usa", "use"]),
    define_domain(word4, [W18, W22, W25, W31, W34, W39, W40, W43, W53, W56, W62,
                          W63, W64, W68, W71, W73, W75, W83, W102, W105, W106,
                          W111, W113, W116, W117, W121, W128, W132, W137, W138,
                          W140, W145, W146],
                          ["ados", "aere", "anel", "apis", "aria", "arno", "bref",
                           "cede", "dada", "demi", "eden", "eole", "epte", "etui",
                           "evoe", "feue", "ille", "ilot", "isar", "lave", "loge",
                           "nato", "nier", "noel", "ouir", "rion", "rode", "rose",
                           "soir", "tera", "trou", "user", "vert"]),
    define_domain(word5, [W3, W4, W5, W9, W16, W21, W23, W26, W27, W44, W47, W52,
                          W65, W66, W69, W72, W79, W86, W87, W91, W93, W98, W99,
                          W100, W107, W109, W125, W130, W133, W139],
                         ["butin", "casse", "cirer", "deite", "doter", "eclos",
                          "ecrin", "emier", "envol", "eolie", "esope", "etier",
                          "evier", "isole", "lieue", "lippe", "mulet", "norme",
```

```
                    "ointe", "peler", "pitre", "raler", "reins", "seine",
                    "timon", "tirer", "tueur", "tulle", "vaine", "valse"]),
    define_domain(word6, [W7, W8, W11, W14, W29, W48, W67, W74, W82, W89, W103,
                    W120, W124, W136, W143],
                   ["aleser", "alliee", "attire", "avilir", "aviser",
                    "blonde", "caisse", "client", "ecrase", "elever",
                    "lacere", "ratier", "rotule", "tavele", "tiroir"]),
    define_domain(word7, [W2, W37, W50, W54, W59, W61, W80, W84, W88, W90, W96,
                    W97, W112, W114, W118, W119, W123],
                   ["apivore", "blocage", "caverne", "colibri", "corsets",
                    "coussin", "ecuries", "egrisee", "eserine", "etoilee",
                    "initier", "notaire", "odieuse", "usuelle", "usurper",
                    "utilise", "voleter"]),
    define_domain(word8, [W1, W6, W20, W35, W36, W42, W51, W110, W115, W135, W144],
                   ["cultiver", "ecervele", "enliasse", "etendard",
                    "fraisier", "gambader", "levrette", "nidifier",
                    "orienter", "savonner", "stimuler"]),
    define_domain(word9, [W13, W17, W19, W104],
                   ["attristee", "elevation", "puissante", "tresorier"]),
    define_domain(word10, [W24, W46, W58, W78],
                   ["etroitesse", "europeenne", "hesperides", "vilipender"]),

% Woerter mit 10 Buchstaben

    all_different([W24, W46, W58, W78]),

% Woerter mit 9 Buchstaben

    all_different([W13, W17, W19, W104]),

% Woerter mit 8 Buchstaben

    all_different([W1, W6, W20, W35, W36, W42, W51, W110, W115, W135, W144]),

% Woerter mit 7 Buchstaben

    all_different([W2, W37, W50, W54, W59, W61, W80, W84, W8, W90, W96, W97,
                   W112, W114, W118, W119, W123]),

% Woerter mit 6 Buchstaben

    all_different([W7, W8, W11, W14, W29, W48, W67, W74, W82, W89, W103,
                   W120, W124, W136, W143]),

% Woerter mit 5 Buchstaben

    all_different([W3, W4, W5, W9, W16, W21, W23, W26, W27, W44, W47, W52,
                   W65, W66, W69, W72, W79, W86, W87, W91, W93, W98, W99,
                   W100, W107, W109, W125, W130, W133, W139]),

% Woerter mit 4 Buchstaben

    all_different([W18, W22, W25, W31, W34, W39, W40, W43, W53, W56, W62,
                   W63, W64, W68, W71, W73, W75, W83, W102, W105, W106,
                   W111, W113, W116, W117, W121, W128, W132, W137, W138,
                   W140, W145, W146]),

% Woerter mit 3 Buchstaben

    all_different([W12, W15, W30, W32, W33, W38, W41, W45, W49, W55, W57,
                   W60, W76, W77, W81, W85, W94, W95, W101, W108, W122,
                   W126, W127, W129, W134, W141, W142]),
```

```
% Woerter mit 2 Buchstaben

    all_different([W10, W28, W70, W92, W131]),

% sameletter - Constraints:

    forward(sameletter(W1, 2, W13, 2)),
    forward(sameletter(W1, 4, W19, 2)),
    forward(sameletter(W1, 6, W26, 2)),
    forward(sameletter(W1, 8, W33, 2)),

    forward(sameletter(W2, 2, W13, 4)),
    forward(sameletter(W2, 4, W19, 4)),
    forward(sameletter(W2, 6, W26, 4)),

    forward(sameletter(W3, 2, W13, 6)),
    forward(sameletter(W3, 4, W19, 6)),
    forward(sameletter(W3, 5, W24, 1)),

    forward(sameletter(W4, 2, W13, 8)),
    forward(sameletter(W4, 4, W19, 8)),
    forward(sameletter(W4, 5, W24, 3)),

    forward(sameletter(W5, 1, W11, 2)),
    forward(sameletter(W5, 3, W17, 2)),
    forward(sameletter(W5, 5, W24, 6)),

    forward(sameletter(W6, 1, W11, 4)),
    forward(sameletter(W6, 3, W17, 4)),
    forward(sameletter(W6, 4, W20, 1)),
    forward(sameletter(W6, 5, W24, 8)),
    forward(sameletter(W6, 6, W27, 1)),
    forward(sameletter(W6, 8, W35, 2)),

    forward(sameletter(W7, 1, W11, 6)),
    forward(sameletter(W7, 2, W14, 1)),
    forward(sameletter(W7, 3, W17, 6)),
    forward(sameletter(W7, 4, W20, 3)),
    forward(sameletter(W7, 5, W24, 10)),
    forward(sameletter(W7, 6, W27, 3)),

    forward(sameletter(W8, 1, W12, 1)),
    forward(sameletter(W8, 2, W14, 3)),
    forward(sameletter(W8, 3, W17, 8)),
    forward(sameletter(W8, 4, W20, 5)),
    forward(sameletter(W8, 5, W25, 1)),
    forward(sameletter(W8, 6, W27, 5)),

    forward(sameletter(W9, 1, W12, 2)),
    forward(sameletter(W9, 2, W14, 4)),
    forward(sameletter(W9, 3, W17, 9)),
    forward(sameletter(W9, 4, W20, 6)),
    forward(sameletter(W9, 5, W25, 2)),

    forward(sameletter(W10, 1, W12, 3)),
    forward(sameletter(W10, 2, W14, 5)),

    forward(sameletter(W15, 1, W14, 2)),
    forward(sameletter(W15, 2, W17, 7)),
    forward(sameletter(W15, 3, W20, 4)),

    forward(sameletter(W16, 1, W14, 6)),
```

```
forward(sameletter(W16, 3, W20, 8)),
forward(sameletter(W16, 4, W25, 4)),
forward(sameletter(W16, 5, W28, 2)),

forward(sameletter(W18, 1, W17, 5)),
forward(sameletter(W18, 2, W20, 2)),
forward(sameletter(W18, 3, W24, 9)),
forward(sameletter(W18, 4, W27, 2)),

forward(sameletter(W21, 1, W19, 7)),
forward(sameletter(W21, 2, W24, 2)),
forward(sameletter(W21, 4, W29, 2)),
forward(sameletter(W21, 5, W34, 3)),

forward(sameletter(W22, 1, W19, 9)),
forward(sameletter(W22, 2, W24, 4)),
forward(sameletter(W22, 4, W29, 4)),

forward(sameletter(W23, 1, W20, 7)),
forward(sameletter(W23, 2, W25, 3)),
forward(sameletter(W23, 3, W28, 1)),
forward(sameletter(W23, 5, W35, 8)),

forward(sameletter(W30, 1, W29, 1)),
forward(sameletter(W30, 2, W34, 2)),
forward(sameletter(W30, 3, W43, 4)),

forward(sameletter(W31, 1, W29, 3)),
forward(sameletter(W31, 2, W34, 4)),
forward(sameletter(W31, 3, W44, 1)),
forward(sameletter(W31, 4, W49, 2)),

forward(sameletter(W32, 1, W29, 5)),
forward(sameletter(W32, 3, W44, 3)),

forward(sameletter(W36, 1, W33, 1)),
forward(sameletter(W36, 4, W54, 1)),
forward(sameletter(W36, 6, W65, 1)),
forward(sameletter(W36, 8, W73, 1)),

forward(sameletter(W37, 1, W34, 1)),
forward(sameletter(W37, 2, W43, 3)),
forward(sameletter(W37, 4, W54, 5)),
forward(sameletter(W37, 6, W65, 5)),
forward(sameletter(W37, 7, W68, 1)),

forward(sameletter(W42, 4, W57, 1)),
forward(sameletter(W42, 5, W62, 2)),
forward(sameletter(W42, 6, W67, 4)),
forward(sameletter(W42, 7, W70, 2)),
forward(sameletter(W42, 8, W75, 3)),

forward(sameletter(W46, 1, W33, 3)),
forward(sameletter(W46, 2, W43, 1)),
forward(sameletter(W46, 4, W54, 3)),
forward(sameletter(W46, 6, W65, 3)),
forward(sameletter(W46, 8, W73, 3)),
forward(sameletter(W46, 9, W81, 2)),
forward(sameletter(W46, 10, W86, 3)),

forward(sameletter(W47, 1, W44, 2)),
forward(sameletter(W47, 2, W49, 3)),
```

```
forward(sameletter(W47, 3, W55, 1)),
forward(sameletter(W47, 4, W61, 3)),
forward(sameletter(W47, 5, W66, 1)),

forward(sameletter(W48, 3, W57, 3)),
forward(sameletter(W48, 4, W62, 4)),
forward(sameletter(W48, 5, W67, 6)),

forward(sameletter(W51, 1, W49, 1)),
forward(sameletter(W51, 2, W54, 7)),
forward(sameletter(W51, 3, W61, 1)),
forward(sameletter(W51, 5, W68, 3)),
forward(sameletter(W51, 6, W74, 1)),
forward(sameletter(W51, 7, W82, 2)),
forward(sameletter(W51, 8, W87, 1)),

forward(sameletter(W52, 1, W50, 2)),
forward(sameletter(W52, 2, W56, 1)),
forward(sameletter(W52, 3, W61, 7)),
forward(sameletter(W52, 4, W66, 5)),
forward(sameletter(W52, 5, W69, 4)),

forward(sameletter(W53, 1, W50, 4)),
forward(sameletter(W53, 2, W56, 3)),
forward(sameletter(W53, 4, W67, 1)),

forward(sameletter(W58, 1, W55, 2)),
forward(sameletter(W58, 2, W61, 4)),
forward(sameletter(W58, 3, W66, 2)),
forward(sameletter(W58, 4, W69, 1)),
forward(sameletter(W58, 5, W74, 4)),
forward(sameletter(W58, 6, W82, 5)),
forward(sameletter(W58, 7, W87, 4)),
forward(sameletter(W58, 8, W92, 2)),
forward(sameletter(W58, 9, W98, 2)),
forward(sameletter(W58, 10, W104, 4)),

forward(sameletter(W59, 1, W55, 3)),
forward(sameletter(W59, 2, W61, 5)),
forward(sameletter(W59, 3, W66, 3)),
forward(sameletter(W59, 4, W69, 2)),
forward(sameletter(W59, 5, W74, 5)),
forward(sameletter(W59, 6, W82, 6)),
forward(sameletter(W59, 7, W87, 5)),

forward(sameletter(W60, 1, W57, 2)),
forward(sameletter(W60, 2, W62, 3)),
forward(sameletter(W60, 3, W67, 5)),

forward(sameletter(W63, 1, W61, 6)),
forward(sameletter(W63, 2, W66, 4)),
forward(sameletter(W63, 3, W69, 3)),
forward(sameletter(W63, 4, W74, 6)),

forward(sameletter(W64, 1, W62, 1)),
forward(sameletter(W64, 2, W67, 3)),
forward(sameletter(W64, 3, W70, 1)),
forward(sameletter(W64, 4, W75, 2)),

forward(sameletter(W71, 1, W68, 4)),
forward(sameletter(W71, 2, W74, 2)),
forward(sameletter(W71, 3, W82, 3)),
```

```
forward(sameletter(W71, 4, W87, 2)),

forward(sameletter(W72, 1, W69, 5)),
forward(sameletter(W72, 3, W83, 2)),
forward(sameletter(W72, 4, W88, 2)),
forward(sameletter(W72, 5, W93, 3)),

forward(sameletter(W76, 1, W73, 2)),
forward(sameletter(W76, 2, W81, 1)),
forward(sameletter(W76, 3, W86, 2)),

forward(sameletter(W77, 1, W73, 4)),
forward(sameletter(W77, 2, W81, 3)),
forward(sameletter(W77, 3, W86, 4)),

forward(sameletter(W78, 1, W74, 3)),
forward(sameletter(W78, 2, W82, 4)),
forward(sameletter(W78, 3, W87, 3)),
forward(sameletter(W78, 4, W92, 1)),
forward(sameletter(W78, 5, W98, 1)),
forward(sameletter(W78, 6, W104, 3)),
forward(sameletter(W78, 7, W108, 3)),
forward(sameletter(W78, 8, W112, 2)),
forward(sameletter(W78, 9, W117, 3)),
forward(sameletter(W78, 10, W124, 1)),

forward(sameletter(W79, 1, W75, 1)),
forward(sameletter(W79, 2, W83, 4)),
forward(sameletter(W79, 3, W88, 4)),
forward(sameletter(W79, 4, W93, 5)),
forward(sameletter(W79, 5, W99, 1)),

forward(sameletter(W80, 1, W75, 4)),
forward(sameletter(W80, 3, W88, 7)),
forward(sameletter(W80, 4, W94, 2)),
forward(sameletter(W80, 5, W99, 4)),
forward(sameletter(W80, 6, W105, 3)),
forward(sameletter(W80, 7, W109, 4)),

forward(sameletter(W84, 1, W83, 1)),
forward(sameletter(W84, 2, W88, 1)),
forward(sameletter(W84, 3, W93, 2)),
forward(sameletter(W84, 4, W98, 5)),
forward(sameletter(W84, 5, W104, 7)),
forward(sameletter(W84, 7, W112, 6)),

forward(sameletter(W85, 1, W83, 3)),
forward(sameletter(W85, 2, W88, 3)),
forward(sameletter(W85, 3, W93, 4)),

forward(sameletter(W89, 1, W86, 1)),
forward(sameletter(W89, 3, W97, 1)),
forward(sameletter(W89, 5, W107, 1)),

forward(sameletter(W90, 1, W86, 5)),
forward(sameletter(W90, 3, W97, 5)),
forward(sameletter(W90, 5, W107, 5)),
forward(sameletter(W90, 6, W111, 3)),
forward(sameletter(W90, 7, W112, 4)),

forward(sameletter(W91, 1, W88, 6)),
forward(sameletter(W91, 2, W94, 1)),
```

```
forward(sameletter(W91, 3, W99, 3)),
forward(sameletter(W91, 4, W105, 2)),
forward(sameletter(W91, 5, W109, 3)),

forward(sameletter(W95, 1, W93, 1)),
forward(sameletter(W95, 2, W98, 4)),
forward(sameletter(W95, 3, W104, 6)),

forward(sameletter(W96, 1, W94, 3)),
forward(sameletter(W96, 2, W99, 5)),
forward(sameletter(W96, 3, W105, 4)),
forward(sameletter(W96, 4, W109, 5)),
forward(sameletter(W96, 6, W118, 7)),
forward(sameletter(W96, 7, W125, 5)),

forward(sameletter(W100, 1, W97, 3)),
forward(sameletter(W100, 3, W107, 3)),
forward(sameletter(W100, 4, W111, 1)),
forward(sameletter(W100, 5, W116, 2)),

forward(sameletter(W101, 1, W97, 7)),
forward(sameletter(W101, 2, W104, 1)),
forward(sameletter(W101, 3, W108, 1)),

forward(sameletter(W102, 1, W98, 3)),
forward(sameletter(W102, 2, W104, 5)),
forward(sameletter(W102, 4, W112, 4)),

forward(sameletter(W103, 1, W99, 2)),
forward(sameletter(W103, 2, W105, 1)),
forward(sameletter(W103, 3, W109, 2)),
forward(sameletter(W103, 5, W118, 4)),
forward(sameletter(W103, 6, W125, 2)),

forward(sameletter(W106, 1, W104, 2)),
forward(sameletter(W106, 2, W108, 2)),
forward(sameletter(W106, 3, W112, 1)),
forward(sameletter(W106, 4, W117, 2)),

forward(sameletter(W110, 1, W107, 4)),
forward(sameletter(W110, 2, W111, 2)),
forward(sameletter(W110, 3, W116, 3)),
forward(sameletter(W110, 5, W128, 4)),
forward(sameletter(W110, 6, W135, 3)),
forward(sameletter(W110, 7, W138, 4)),
forward(sameletter(W110, 8, W143, 3)),

forward(sameletter(W113, 1, W112, 3)),
forward(sameletter(W113, 2, W117, 4)),
forward(sameletter(W113, 3, W124, 2)),
forward(sameletter(W113, 4, W130, 1)),

forward(sameletter(W114, 1, W112, 5)),
forward(sameletter(W114, 3, W124, 4)),
forward(sameletter(W114, 4, W130, 3)),
forward(sameletter(W114, 5, W136, 2)),
forward(sameletter(W114, 7, W144, 1)),

forward(sameletter(W115, 1, W112, 7)),
forward(sameletter(W115, 2, W118, 1)),
forward(sameletter(W115, 3, W124, 6)),
forward(sameletter(W115, 4, W130, 5)),
```

```
forward(sameletter(W115, 5, W136, 4)),
forward(sameletter(W115, 7, W144, 3)),

forward(sameletter(W119, 1, W116, 1)),
forward(sameletter(W119, 3, W128, 2)),
forward(sameletter(W119, 4, W135, 1)),
forward(sameletter(W119, 5, W138, 2)),
forward(sameletter(W119, 6, W143, 1)),

forward(sameletter(W120, 1, W117, 1)),
forward(sameletter(W120, 3, W129, 2)),
forward(sameletter(W120, 4, W135, 6)),
forward(sameletter(W120, 5, W139, 2)),
forward(sameletter(W120, 6, W143, 6)),

forward(sameletter(W121, 1, W118, 3)),
forward(sameletter(W121, 2, W125, 1)),
forward(sameletter(W121, 4, W136, 6)),

forward(sameletter(W122, 1, W118, 5)),
forward(sameletter(W122, 2, W125, 3)),
forward(sameletter(W122, 3, W131, 1)),

forward(sameletter(W123, 1, W118, 6)),
forward(sameletter(W123, 2, W125, 4)),
forward(sameletter(W123, 3, W131, 2)),
forward(sameletter(W123, 5, W140, 3)),
forward(sameletter(W123, 6, W144, 8)),
forward(sameletter(W123, 7, W146, 3)),

forward(sameletter(W126, 1, W124, 3)),
forward(sameletter(W126, 2, W130, 2)),
forward(sameletter(W126, 3, W136, 1)),

forward(sameletter(W127, 1, W124, 5)),
forward(sameletter(W127, 2, W130, 4)),
forward(sameletter(W127, 3, W136, 3)),

forward(sameletter(W132, 1, W128, 3)),
forward(sameletter(W132, 2, W135, 2)),
forward(sameletter(W132, 3, W138, 3)),
forward(sameletter(W132, 4, W143, 2)),

forward(sameletter(W133, 1, W129, 1)),
forward(sameletter(W133, 2, W135, 5)),
forward(sameletter(W133, 3, W139, 1)),
forward(sameletter(W133, 4, W143, 5)),

forward(sameletter(W134, 1, W129, 3)),
forward(sameletter(W134, 2, W135, 7)),
forward(sameletter(W134, 3, W139, 3)),

forward(sameletter(W137, 1, W135, 8)),
forward(sameletter(W137, 2, W139, 4)),
forward(sameletter(W137, 4, W145, 2)),

forward(sameletter(W141, 1, W140, 1)),
forward(sameletter(W141, 2, W144, 6)),
forward(sameletter(W141, 3, W146, 1)),

forward(sameletter(W142, 1, W140, 2)),
forward(sameletter(W142, 2, W144, 7)),
```

```
    forward(sameletter(W142, 3, W146, 2)),

    writeln('Starting instantiating variables now!'),
    writeln('Instantiating word10'),
    instantiate([W24, W46, W58, W78]),
    writeln('instantiating word9'),
    instantiate([W13, W17, W19, W104]),
    writeln('instantiating word8'),
    instantiate([W1, W6, W20, W35, W36, W42, W51, W110, W115, W135, W144]),
    writeln('instantiating word7'),
    instantiate([W2, W37, W50, W54, W59, W61, W80, W84, W8, W90, W96, W97,
                 W112, W114, W118, W119, W123]),
    writeln('instantiating word6'),
    instantiate([W7, W8, W11, W14, W29, W48, W67, W74, W82, W89,
                  W103, W120, W124, W136, W143]),
    writeln('instantiating word5'),
    instantiate([W3, W4, W5, W9, W16, W21, W23, W26, W27, W44, W47, W52, W65,
                 W66, W69, W72, W79, W86, W87, W91, W93, W98, W99, W100, W107,
                 W109, W125, W130, W133, W139]),
    writeln('instantiating word4 '),
    instantiate([W18, W22, W25, W31, W34, W39, W40, W43, W53, W56, W62,
                 W63, W64, W68, W71, W73, W75, W83, W102, W105, W106, W111,
                 W113, W116, W117, W121, W128, W132, W137, W138, W140, W145,
                 W146]),
    writeln('instantiating word3 '),
    instantiate([W12, W15, W30, W32, W33, W38, W41, W45, W49, W55, W57,
                  W60, W76, W77, W81, W85, W94, W95, W101, W108, W122,
                  W126, W127, W129, W134, W141, W142]),
    writeln('Instantiating word2 '),
    instantiate([W10, W28, W70, W92, W131]).

all_different([]).
all_different([X|Y]) :-
    out_of(X, Y),
    all_different(Y).

out_of(_, []).
out_of(X, [Y|Z]) :-
    forward(X \= Y),
    out_of(X, Z).

sameletter(W1, P1, W2, P2) :-
     substring(W1, P1, 1, Letter),
     substring(W2, P2, 1, Letter).
```

A.2 The Map-Colouring Example

In the following, a FIDO-II program colouring the map of Europe is displayed. See chapter 7.2 for a description of the problem that can be regarded as instance of graph-colouring problems.

```
map([Norway,Sweden,Danmark,Finland,Russia,Poland,CSFR,Romania,Hungary,
     Bulgaria,Greece,Albania,Turkey,Germany,Austria,Italy,Switzerland,
     Luxemburg,Belgium,Netherlands,Spain,France,Croatia,Serbia,Slovenia,
     Bosnia, Macedonia,Montenegro,Scotland,Wales,England,Ireland,Ulster,
     Iceland, Portugal]) :-
    define_domain(colours,[Norway,Sweden,Danmark,Finland,Russia,Poland,CSFR,
                           Romania,Hungary, Bulgaria,Greece,Albania,Turkey,
                           Germany,Austria,Italy,Switzerland, Luxemburg,
```

```
                    Belgium,Netherlands,Spain,France,Croatia,Serbia,
                    Slovenia, Bosnia, Macedonia,Montenegro,Scotland,
                    Wales,England,Ireland,Ulster, Iceland, Portugal],
              [1, 2, 3, 4]),

forward(Norway \= Sweden),
forward(Norway \= Finland),
forward(Norway \= Iceland),
forward(Norway \= Scotland),
forward(Sweden \= Finland),
forward(Sweden \= Danmark),
forward(Danmark \= Germany),
forward(Danmark \= England),
forward(Finland \= Russia),
forward(Russia \= Poland),
forward(Russia \= CSFR),
forward(Russia \= Hungary),
forward(Russia \= Romania),
forward(Poland \= Germany),
forward(Poland \= CSFR),
forward(CSFR \= Germany),
forward(CSFR \= Hungary),
forward(CSFR \= Austria),
forward(Romania \= Hungary),
forward(Romania \= Bulgaria),
forward(Romania \= Serbia),
forward(Hungary \= Austria),
forward(Hungary \= Serbia),
forward(Hungary \= Slovenia),
forward(Hungary \= Croatia),
forward(Bulgaria \= Greece),
forward(Bulgaria \= Macedonia),
forward(Bulgaria \= Turkey),
forward(Bulgaria \= Serbia),
forward(Greece \= Albania),
forward(Greece \= Turkey),
forward(Greece \= Macedonia),
forward(Albania \= Montenegro),
forward(Albania \= Macedonia),
forward(Albania \= Italy),
forward(Germany \= Austria),
forward(Germany \= Switzerland),
forward(Germany \= France),
forward(Germany \= Luxemburg),
forward(Germany \= Belgium),
forward(Germany \= Netherlands),
forward(Austria \= Switzerland),
forward(Austria \= Slovenia),
forward(Italy \= Slovenia),
forward(Italy \= France),
forward(Italy \= Switzerland),
forward(Switzerland \= France),
forward(Luxemburg \= France),
forward(Luxemburg \= Belgium),
forward(Belgium \= Netherlands),
forward(Belgium \= England),
forward(Netherlands \= England),
forward(Spain \= France),
forward(Spain \= Portugal),
forward(Spain \= Ireland),
forward(France \= England),
forward(Croatia \= Slovenia),
```

```
    forward(Croatia \= Bosnia),
    forward(Croatia \= Serbia),
    forward(Serbia \= Bosnia),
    forward(Serbia \= Macedonia),
    forward(Serbia \= Montenegro),
    forward(Serbia \= Slovenia),
    forward(Bosnia \= Montenegro),
    forward(Macedonia \= Montenegro),
    forward(Scotland \= England),
    forward(Scotland \= Ulster),
    forward(Scotland \= Iceland),
    forward(Wales \= England),
    forward(Wales \= Ireland),
    forward(Wales \= Ulster),
    forward(England \=Ireland),
    forward(England \= Ulster),
    forward(Ireland \= Ulster),
    forward(Ireland \= Iceland),
    instantiate([Norway,Sweden,Danmark,Finland,Russia,Poland,CSFR,Romania,
                 Hungary, Bulgaria,Greece,Albania,Turkey,Germany,Austria,Italy,
                 Switzerland, Luxemburg,Belgium,Netherlands,Spain,France,
                 Croatia,Serbia,Slovenia, Bosnia, Macedonia,Montenegro,
                 Scotland,Wales,England,Ireland,Ulster, Iceland, Portugal]).
```

A.3 The Scheduling Example

In the following, the FIDO-II source code for the scheduling problem of chapter 7.3 is listed.

```
/* scheduling example. The shifts  are mapped to the integers from 1 to 9. */
/* Each employee has five shifts per week, which are defined by the five   */
/* variables XXX1 to XXX5 for each employee XXX.                            */

schedule( [Harry1,Sally1, Fred1, Isabel1, Paul1, Pamela1, Tina1, Anna1, Mary1,
           Harry2,Sally2, Fred2, Isabel2, Paul2, Pamela2, Tina2, Anna2, Mary2,
           Harry3,Sally3, Fred3, Isabel3, Paul3, Pamela3, Tina3, Anna3, Mary3 ]) :-
    define_domain(shifts, [Harry1,Sally1, Fred1, Isabel1, Paul1, Pamela1,
                           Tina1, Anna1, Mary1, Harry2,Sally2, Fred2, Isabel2,
                           Paul2, Pamela2, Tina2, Anna2, Mary2,Harry3,Sally3,
                           Fred3, Isabel3, Paul3, Pamela3, Tina3, Anna3, Mary3 ],
                  1..9),
\* Constraint 1 : *\
    not_subsequent([Harry1, Harry2, Harry3]),
    not_subsequent([Sally1, Sally2, Sally3]),
    not_subsequent([Fred1, Fred2, Fred3]),
    not_subsequent([Isabel1, Isabel2, Isabel3]),
    not_subsequent([Paul1, Paul2, Paul3]),
    not_subsequent([Pamela1, Pamela2, Pamela3]),
    not_subsequent([Tina1, Tina2, Tina3]),
    not_subsequent([Anna1, Anna2, Anna3]),
    not_subsequent([Mary1, Mary2, Mary3]),
\*  Constraint number 9 *\
    forward(Harry1 \= 3),
    forward(Harry2 \= 6),
    forward(Harry3 \= 9),
\* Constraint number 10 *\
    forward(Tina1 \= 1),
    forward(Tina2 \= 4),
```

```
    forward(Tina3 \= 7),
\* Constraint number 6 *\
    forward(Pamela1 =\= Paul1 + 1),
    forward(Pamela1 =\= Paul1 - 1),
    forward(Pamela1 \= Paul1),
    forward(Pamela2 =\= Paul2 + 1),
    forward(Pamela2 =\= Paul2 - 1),
    forward(Pamela2 \= Paul2),
    forward(Pamela3 =\= Paul3 + 1),
    forward(Pamela3 =\= Paul3 - 1),
    forward(Pamela3 \= Paul3),

\* Constraint number 5 *\
    forward(Fred1 \= Isabel1),
    forward(Fred2 \= Isabel2),
    forward(Fred3 \= Isabel3),

\* Constraint number 7 *\
    forward(Tina1 \= Mary1),
    forward(Tina2 \= Mary2),
    forward(Tina3 \= Mary3),

\* Constraint number 12 *\
    forward(Fred1 = Pamela1),
    forward(Fred2 = Pamela2),
    forward(Fred3 = Pamela3),

\* Constraint number 4 *\
    forward(Harry1 = Sally1),
    forward(Harry2 = Sally2),
    forward(Harry3 = Sally3),

\*  Constraint number 8 *\
    forward(Anna1 =:= Paul1 - 1),
    forward(Anna2 =:= Paul2 - 1),
    forward(Anna3 =:= Paul3 - 1),

\* constraint number 11 *\
    forward(Mary1 = 2),
    forward(Mary2 = 5),
    forward(Mary3 = 8),

    instantiate_card([Harry1,Sally1, Fred1, Isabel1, Paul1, Pamela1, Tina1, Anna1, Mary1,
                      Harry2, Sally2, Fred2, Isabel2, Paul2, Pamela2, Tina2, Anna2, Mary2,Harry3,
                      Sally3, Fred3, Isabel3, Paul3, Pamela3, Tina3, Anna3, Mary3 ],
 9, 3).



not_subsequent([]).
not_subsequent([H]).
not_subsequent([H|T]) :-
    not_subsequent_aux(H, T),
    not_subsequent(T).

not_subsequent_aux(El, []).
not_subsequent_aux(El, [H|T]) :-
    forward(El =\= H - 1),
    forward(El < H),
    not_subsequent_aux(El, T).

empty_intersection([], _).
```

```
empty_intersection([H|T], List) :-
    out_of(H, List),
    empty_intersection(T, List).

out_of(_, []).
out_of(X, [H|T]) :-
    forward(X \= H),
    out_of(X, T).

in_between([], _, _).
in_between([H|T], From, To) :-
    forward(H >= From),
    forward(H =< To),
    in_between(T, From, To).
```

Appendix B

Implementation Issues

In the following a listing is given of the files used by and needed for FIDO-II. From a logical point of view, the code can be divided into the following portions:

1. The FIDO main shell program which starts precompilation of an input file.
2. The code needed for the first preprocessing scan.
3. The code required for the second preprocessing scan.
4. The code for the third preprocessing scan.
5. The normalizer code and the code for the redefinition of user-defined constraints.
6. The database facts and declarations.
7. The code for stream output.
8. A domain variable predicates library.
9. A control library containing the definitions of the consistency algorithms.
10. A library providing utility function for open lists.
11. Diverse utility functions.
12. The redefinitions of the built-in constraints.

B.1 The Code Portions

- **/home/jmueller/clp/transform/fido.pl:** The file contains the PROLOG implementation of the FIDO-II main program. It embodies the top-level of the FIDO-II preprocessor and contains the calls to the specific preprocessing scans.
- **/home/jmueller/clp/transform/process_s1.pl:** This file contains the source code for the first scan of the input program.
- **/home/jmueller/clp/transform/process_s2.pl:** The code implementing the second preprocessing phase, which performs the handling of the domains, can be found in this file.

- **/home/jmueller/clp/transform/process_s3.pl:** This file contains the implementation of phase three of the FIDO-II preprocessor. It performs the changes necessary in the source code and handles the consistency declarations.
- **/home/jmueller/clp/transform/normalize.pl, explode.pl:** This file contains the code defining the constraint normalizer and the code generator of the redefinitions of user-defined constraints.
- **/home/jmueller/clp/lib/clplib_ff.pl:** Here, the FIDO library predicates providing facilities for the creation and the maintaining of domain variables at run-time and the instantiation built-ins are contained.
- **/home/jmueller/clp/lib/utils.pl:** This file consists of several helpful predicates used by other FIDO-II predicates.
- **/home/jmueller/clp/lib/openlists.pl:** This is a library which provides important predicates for handling open lists which are represented as lists whose last element is a variable. For example, the list $[1, 2, 3|Var]$ is an open list. The library contains operations such as membership and set operations on open lists.
- **/home/jmueller/clp/control:** This directory contains the forward-checking algorithms working on built-in and user-defined constraints.
- **/home/jmueller/clp/redef:** This directory contains the four sub-directories **eq**, **ne**, **gt**, and **lt** which provide the constraint redefinition of the equality, inequality respectively ordering built-in constraints.
- **/home/jmueller/clp/transform/etc.pl, declarations.pl:** These two files contain the database facts characterizing the built-in constraint and the declarations of dynamic predicates and of the operators used by FIDO-II.

Known Restrictions

In this section, some known restrictions of the current FIDO-II prototype version are listed. The listing contains some details mentioned in the work, but also some new issues interesting for the programmer.

- The maximal number of arguments permitted for user-defined constraints is five.
- The implicit unification of domain variables, i.e. by clause-head unification is forbidden. It is *not* automatically recognized by the system and will lead to unpredictable behaviour of the program.
- First-fail heuristics using the number of constraints of a domain-variable is not implemented.
- The FIDO-II built-in constraints which are available in the current version are $= /2, =:= /2, \backslash = /2, = \backslash = /2, > /2$, and $< /2$.
- The use of some SEPIA library predicates, which are not regarded as built-ins by SEPIA (i.e. the **is_built_in** /1 predicate fails), but which are no user-defined predicates either, must be added to the definition of the FIDO **built_in** /1 predicate which is defined in **/home/jmueller/clp/lib/utils.pl'**.

- For big examples such as the crossword puzzle program or n queens with $n \geq 64$, the global and/or local PROLOG heap must be resized. This can be achieved by starting SEPIA with the command:

  sepia -g n_1 **-l** n_2[1]

  n_1 and n_2 should be chosen sufficiently big. For most applications, $n_1 = 5000$ and $n_2 = 2000$ should suffice.

- For reasons of time, the program code implementing weak looking-ahead does not have "product niveau". It is not thoroughly tested. Thus, its correctness cannot be asserted even if soundness and completeness of the algorithm itself have been proved in this work.

[1] I *do* know exactly that somebody is going to type "sepia -g n_1 -l n_2" right now!

Bibliography

[ADH91] A. Abecker, D. Drollling, and P. Hanschke. Taxon: A concept language with concrete domains. In Michael M. Richter and Harold Boley, editors, *Preprints of the Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91)*. DFKI (German Research Center for Artificial Intelligence), June 1991.

[AkN86] H. Aït-kaci and R. Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, October 1986.

[BFL83] R.J. Brachman, R.E. Fikes, and H.J. Levesque. Krypton: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, October 1983.

[BM72] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BM73] G. Battani and H. Meloni. *Interpreteur du Language de Programmation Prolog.* Technical Report, Groupe Intelligence Artificielle, Universite Aix-Marseille, 1973.

[BMMW89] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proc. of ICLP 89*, pages 149–164, 1989.

[Bor79] A. Borning. Thinglab - a constraint-oriented simulation laboratory. Technical Report SSL-79-3, Xerox PARC, Palo Alto, California, July 1979.

[Bor85a] A. Borning. Constraints and functional programming. Technical Report 85-09-05, University of Washington, Computer Science Dept., September 1985.

[Bor85b] A. Borning. Defining constraints graphically. Technical Report 85-09-05, University of Washington, Computer Science Dept., September 1985.

[Bra86] I. Bratko. *PROLOG Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[Bru78] M. Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In *Colloquium on Mathematical Logic in Programming, Salgotarjan(Hungary)*, pages 215–258, 1978.

[Bru81] M. Bruynooghe. Solving combinatorical search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[Car87] M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987. MIT Press.

[Cha81] D.D. Chamberlin. Support for repetitive transactions and ad-hoc queries in system r. *ACM Trans. Database Systems*, 6(1):70–94, 1981.

[CKv83] A. Colmerauer, H. Kanoui, and M. vanCaneghem. Prolog, bases theoretiques et developpements. *Techniques et Sciences Informatiques*, 2(4):271–311, 1983.

[CM79] K.L. Clark and F. McCabe. The control facilities of ic-prolog. In *Expert Systems in the Micro Electronic Age*, pages 122–149. Edinburgh University Press, 1979.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[Coh83] A.G. Cohn. Improving the expressiveness of many sorted logic. In *Proceedings of AAAI-83, Washington, D.C.*, pages 84–87, 1983.

[Col82] A. Colmerauer. PROLOG II Manual de Reference et Model Theoretique, Mærz 1982.

[Col87a] A. Colmerauer. Introduction to Prolog III. In *ESPRIT '87*, pages 611–629. North Holland, 1987.

[Col87b] A. Colmerauer. Opening the Prolog III Universe. *BYTE*, pages 177–182, August 1987.

[Dan63] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[DL84] M. Dincbas and J.P. Lepape. Metacontrol of logic programs in metalog. In *Proc. of the International Conference on Fifth Genration Computer Systems(FGCS84)*, pages 361–370. ICOT, Tokyo, Japan, November 1984.

[dSPRB90] D. de Schreye, D. Pollet, J. Ronsyn, and M. Bruynooghe. Implementing Finite–domain Constraint Logic Programming on Top of a PROLOG–System with Delay–mechanism. In N. Jones, editor, *Proc. of ESOP 90*, pages 106–117, 1990.

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proc. FGCS'88*, Tokyo, Japan, December 1988.

[DvHS+89] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, and A. Herold. The CHIP System: Constraint Handling In Prolog. pages 774–775, 1989.

[ED89] ed. E. Dechter. Proceedings of Workshop on Constraint Solving, IJCAI89, Detroit, 1989.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Fag79] R. Fagin. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Systems*, 4(3):387–415, 1979.

[GL80] H. Gallaire and C. Lasserre. A control metalanguage for logic programming. In *Proc. of Logic Programming Workshop, Debrecan, Hungary*, pages 73–79, July 1980.

[Hä87] T. Härder. *Realisierung von operationalen Schnittstellen.* Springer, Berlin, 1987.

[HB78] P.H. Howard and K.W. Borgendale. System/38 machine indexing support. *IBM system/38 Technical Developments*, pages 67–69, 1978.

[HE80] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for csp's. *Artifiacial Intelligence*, 14(3), 1980.

[Hei91] H.-G. Hein. Consistency Techniques in WAM-based Architectures. Diploma thesis, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, 1991. Forthcoming.

[Hol90a] C. Holzbaur. Integration of extended unification into a prolog interpreter. Technical Report 90-9, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, 1990.

[Hol90b] Ch. Holzbaur. Realization of Forward Checking in Logic Programming through Extended Unification. Technical Report TR–90–11, Austrian Research Institute for Artifical Intelligence, June 1990.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. POPL-87*, Munich, Germany, 1987.

[JLM86] J. Jaffar, J.L. Lassez, and M.J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Relations, Functions and Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[JM87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proc. of ICLP 87*, pages 196–218, 1987.

[JMSY90] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. Technical Report CMU–CS–90–181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1990.

[KJ84] M. Konopasek and S. Jayaraman. *The TK!Solver Book*. Osborne/McGraw-Hill, Berkeley, California, 1984.

[Knu75] D.E. Knuth. *Sorting and Searching: The Art of Computer Programming*. Addison-Wesley, New York, 1975.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In J. Rosenfeld, editor, *Information Processing 74*, pages 556–574. North Holland, Amsterdam, 1974.

[Kow79] R. Kowalski. Algorithm = Logic + Control. *Journal of the ACM*, 22:424–436, 1979.

[Lau78] J.L. Lauriere. A language and a program for stating and solving combinatorical problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[Mac77] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[MAC+89] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D.H. de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA – An Extendible Prolog System. In G. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress*, pages 1127–1132, August 1989.

[Meh75] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4):287–295, 1975.

[MHM91] M. Meyer, H.-G. Hein, and J. Müller. FIDO: Finite Domain Consistency Techniques in Logic Programming. In *Proceedings of the 2nd Russian Conference on Logic Programming.* Lecture Notes in Artificial Intelligence, Springer-Verlag, Heidelberg, 1991.

[ML75] W.D. Maurer and T.G. Lewis. Hash table methods. *ACM Computing Survey*, 7(1):5–19, 1975.

[MM77a] J.R. McSkimin and J. Minker. A predicate calculus based semantic network for question-answering systems. Technical Report TR-509, Dept. of Comp.Science, University of Maryland, March 1977.

[MM77b] J.R. McSkimin and J. Minker. The use of a semantic network in a deductive question-answering system. Technical Report TR-506, Dept. of Comp.Science, University of Maryland, February 1977.

[Mor68] D.R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal ACM*, 15(4):514–534, 1968.

[Nai82] L. Naish. An introduction to mu-prolog. Technical Report DCS-TR-164, Dept. of Computer Science, University of Melbourne, AUS, 1982.

[Nai85] L. Naish. Automating control for logic programs. *Journal of Logic Programming*, 2(3):167–184, October 1985.

[Nel84] G. Nelson. How to use juno. Manuscript CGN11, Computer Science Laboratory, Xerox PARC, Palo Alto, California, 1984.

[Nie74] J. Nievergelt. Binary search trees and file organization. *ACM Computing Survey*, 6(3):195–207, 1974.

[RB91] Michael M. Richter and Harold Boley, editors. *Preprints of the Proceedings of the International Workshop on Processing Declarative Knowledge (PDK'91).* DFKI (German Research Center for Artificial Intelligence), June 1991.

[Ric78] M. M. Richter. *Logikkalküle.* Teubner Studienbücher Informatik, 1978.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Rou75] P. Roussel. *PROLOG Manual de Reference et d'Utilisation*, 1975.

[Sch91] S. Schrödl. FIDO: Ein Constraint-Logic-Programming-System mit Finite Domains. ARC-TEC Discussion Paper 91-05, DFKI GmbH, Postfach 2080, D-6750 Kaiserslautern, June 1991.

[SEP90] *SEPIA 3.0 User Manual*, Juni 1990.

[SEP91] *SEPIA 3.0.16 User Manual*, 1991.

[SS77] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, pages 135–196, September 1977.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Massachusetts, 1986.

[Ste80] G.L. Steele. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, MIT, Cambridge, Massachusettes, August 1980.

[Sut63] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *IFIPS Proceedings of the Spring Joint Computer Conference*, January 1963.

[Tay91] A. Taylor. *High Performance PROLOG Implementation.* PhD thesis, Basser Dpt. of Computer Science, University of Sydney, AU, 1991.

[Tv89] O. Thibault and P. van Hentenryck. Implementation of chip on kcm. Intermediate Report, November 1989.

[TZ88] J. Thom and J. Zobel. *NU-Prolog Reference Manual, Version 1.3.* Technical Report 86-10, Department of Computer Science, University of Melbourne, Melbourne, Australien, 1988.

[van87a] P. van Hentenryck. A Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, Milan, Italy, August 1987.

[van87b] P. van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur, Belgium, July 1987.

[van89a] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, Ma., 1989.

[van89b] P. van Hentenryck. Parallel Constraint Satisfaction in Logic Programming (Preliminary results of CHIP within PEPSys). In *Proc. of ICLP 89*, pages 165–180, 1989.

[van91] P. van Hentenryck. Constraint logic programming. Talk at PDK 91, Kaiserslautern Germany, 1991.

[Vod88] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical Report, Complete Logic Systems, North Vancouver, British Columbia, Canada, 1988.

[vW80] C.J. van Wyk. *A Language for Typesetting Graphics.* PhD thesis, Stanford University, Palo Alto, California, June 1980.

[Wal84a] C. Walther. A mechanical solution to shubert's steamroller by many sorted resolution. In *4th National Conference on AI (AAAI-84), Austin, Texas*, 1984.

[Wal84b] C. Walther. Unification in many sorted theories. In *Proceedings of the 6th European Conference on AI, Pisa (Italy)*, September 1984.

[War83] D. H. D. Warren. *An Abstract Prolog Instruction Set.* Technical Note 309, SRI International, Menlo Park, California, 1983.

[Wed74] H. Wedekind. On the selection of access paths in a database system. In *Klimbie, Koffeman (eds.): Database Management*, pages 385–397. North Holland, 1974.

[Win75] T. Winograd. Frame representations and the declarative / procedural controversy. In D.G. Bobrow and A. Collins, editors, *Representation and Understanding - Studies in Cognitive Science*, pages 185–210. Academic Press, 1975.

[Wir75] N. Wirth. *Algorithmen und Datenstrukturen.* Teubner, Stuttgart, 1975.

[WRCS87] H. Westphal, P. Robert, J. Chassin, and J.C. Syre. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *International Symposium on Logic Programming*, pages 436–448, September 1987.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-90-03
Andreas Dengel, Nelson M. Mattos: Integration of Document Representation, Processing and Management
18 pages

RR-90-04
Bernhard Hollunder, Werner Nutt: Subsumption Algorithms for Concept Languages
34 pages

RR-90-05
Franz Baader: A Formal Definition for the Expressive Power of Knowledge Representation Languages
22 pages

RR-90-06
Bernhard Hollunder: Hybrid Inferences in KL-ONE-based Knowledge Representation Systems
21 pages

RR-90-07
Elisabeth André, Thomas Rist: Wissensbasierte Informationspräsentation:
Zwei Beiträge zum Fachgespräch Graphik und KI:
1. Ein planbasierter Ansatz zur Synthese illustrierter Dokumente
2. Wissensbasierte Perspektivenwahl für die automatische Erzeugung von 3D-Objektdarstellungen

24 Seiten

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-09
Susanne Biundo: Plan Generation Using a Method of Deductive Program Synthesis
17 pages

RR-90-10
Franz Baader, Hans-Jürgen Bürckert, Bernhard Hollunder, Werner Nutt, Jörg H. Siekmann: Concept Logics
26 pages

RR-90-11
Elisabeth André, Thomas Rist: Towards a Plan-Based Synthesis of Illustrated Documents
14 pages

RR-90-12
Harold Boley: Declarative Operations on Nets
43 pages

RR-90-13
Franz Baader: Augmenting Concept Languages by Transitive Closure of Roles: An Alternative to Terminological Cycles
40 pages

RR-90-14
Franz Schmalhofer, Otto Kühn, Gabriele Schmidt: Integrated Knowledge Acquisition from Text, Previously Solved Cases, and Expert Memories
20 pages

RR-90-15
Harald Trost: The Application of Two-level Morphology to Non-concatenative German Morphology
13 pages

RR-90-16
Franz Baader, Werner Nutt: Adding Homomorphisms to Commutative/Monoidal Theories, or: How Algebra Can Help in Equational Unification
25 pages

RR-90-17
Stephan Busemann: Generalisierte Phasenstrukturgrammatiken und ihre Verwendung zur maschinellen Sprachverarbeitung
114 Seiten

RR-91-01
Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, Gert Smolka: On the Expressivity of Feature Logics with Negation, Functional Uncertainty, and Sort Equations
20 pages

RR-91-02
Francesco Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, Werner Nutt: The Complexity of Existential Quantification in Concept Languages
22 pages

RR-91-03
B.Hollunder, Franz Baader: Qualifying Number Restrictions in Concept Languages
34 pages

RR-91-04
Harald Trost: X2MORF: A Morphological Component Based on Augmented Two-Level Morphology
19 pages

RR-91-05
Wolfgang Wahlster, Elisabeth André, Winfried Graf, Thomas Rist: Designing Illustrated Texts: How Language Production is Influenced by Graphics Generation.
17 pages

RR-91-06
Elisabeth André, Thomas Rist: Synthesizing Illustrated Documents A Plan-Based Approach
11 pages

RR-91-07
Günter Neumann, Wolfgang Finkler: A Head-Driven Approach to Incremental and Parallel Generation of Syntactic Structures
13 pages

RR-91-08
Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist: WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

RR-91-09
Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta: RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

RR-91-10
Franz Baader, Philipp Hanschke: A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

RR-91-11
Bernhard Nebel: Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, Stanley Peters: The Absorption Principle and E-Type Anaphora
33 pages

RR-91-13
Gert Smolka: Residuation and Guarded Rules for Constraint Logic Programming
17 pages

RR-91-14
Peter Breuer, Jürgen Müller: A Two Level Representation for Spatial Relations, Part I
27 pages

RR-91-15
Bernhard Nebel, Gert Smolka: Attributive Description Formalisms ... and the Rest of the World
20 pages

RR-91-16
Stephan Busemann: Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

RR-91-17
Andreas Dengel, Nelson M. Mattos: The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

RR-91-18
John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein: A Diagnostic Tool for German Syntax
20 pages

RR-91-19
Munindar P. Singh: On the Commitments and Precommitments of Limited Agents
15 pages

RR-91-20
Christoph Klauck, Ansgar Bernardi, Ralf Legleitner FEAT-Rep: Representing Features in CAD/CAM
48 pages

RR-91-21
Klaus Netter: Clause Union and Verb Raising Phenomena in German
38 pages

RR-91-22
Andreas Dengel: Self-Adapting Structuring and Representation of Space
27 pages

RR-91-23
Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-24
Jochen Heinsohn: A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder: Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

RR-91-26
M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger: Integrated Plan Generation and Recognition
- A Logic-Based Approach -
17 pages

RR-91-27
A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer: ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

RR-91-28
Rolf Backofen, Harald Trost, Hans Uszkoreit: Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

RR-91-29
Hans Uszkoreit: Strategies for Adding Control Information to Declarative Grammars
17 pages

RR-91-30
Dan Flickinger, John Nerbonne: Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39pages

RR-91-31
H.-U. Krieger , J. Nerbonne: Feature-Based Inheritance Networks for Computational Lexicons
11 pages

RR-91-32
Rolf Backofen, Lutz Euler, Günther Görz: Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

RR-91-33
Franz Baader , Klaus Schulz: Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

RR-91-35
Winfried Graf, Wolfgang Maaß: Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

DFKI Technical Memos

TM-90-04
Franz Baader, Hans-Jürgen Bürckert, Jochen Heinsohn, Bernhard Hollunder, Jürgen Müller, Bernhard Nebel, Werner Nutt, Hans-Jürgen Profitlich: Terminological Knowledge Representation: A Proposal for a Terminological Logic
7 pages

TM-91-01
Jana Köhler: Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

TM-91-02
Knut Hinkelmann: Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

TM-91-03
Otto Kühn, Marc Linster, Gabriele Schmidt: Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

TM-91-04
Harold Boley (Ed.): A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel , Rainer Bleisinger: Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

TM-91-06
Johannes Stein: Aspects of Cooperating Agents
22 pages

TM-91-08
Munindar P. Singh: Social and Psychological Commitments in Multiagent Systems
11 pages

TM-91-09
Munindar P. Singh: On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

TM-91-10
Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch: Tree Adjoining Grammars mit Unifikation
149 pages

TM-91-11
Peter Wazinski: Generating Spatial Descriptions for Cross-modal References
21 pages

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann:
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

DFKI Documents

D-90-06
Andreas Becker: The Window Tool Kit
66 Seiten

D-91-01
Werner Stein , Michael Sintek: Relfun/X - An Experimental Prolog Implementation of Relfun
48 pages

D-91-02
Jörg P. Müller: Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining
127 pages

D-91-03
Harold Boley, Klaus Elsbernd, Hans-Günther Hein, Thomas Krause: RFM Manual: Compiling RELFUN into the Relational/Functional Machine
43 pages

D-91-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

D-91-06
Gerd Kamp: Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

D-91-07
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

D-91-08
Thomas Krause: Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 pages

D-91-09
David Powers and Lary Reeker (Eds):
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-10
Donald R. Steiner, Jürgen Müller (Eds.):
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-91-11
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

D-91-12
Bernd Bachmann:
HieraCon - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

D-91-13
International Workshop on Terminological Logics
Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck
131 pages

D-91-14
Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS: Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

Design and Implementation of a Finite Domain Constraint Logic Programming System based on PROLOG with Coroutining

Jörg P. Müller

D-91-02
Document