Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-91-12

HieraCon

A Knowledge Representation System with Typed Hierarchies and Constraints

Bernd Bachmann

August 1991

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

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

Stuhlsatzenhausweg 3
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 by the shareholder companies ADV/Orga, AEG, IBM, Insiders, Fraunhofer Gesellschaft, GMD, Krupp-Atlas, Mannesmann-Kienzle, Philips, 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

HieraCon
A Knowledge Representation System with Typed Hierarchies and Constraints

Bernd Bachmann

DFKI-D-91-12

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

HieraCon

A Knowledge Representation System with
Typed Hierarchies and Constraints

Bernd Bachmann

August 1991

Preface

This project evolved while I joined the Knowledge Based Programming Department (KBPD) at the Hewlett–Packard Laboratories in Bristol, England. Throughout these three months the feasibility of a constraint–based configuration task were investigated and lead to a first approach for a configuration system for HP–workstations. Back at the German Research Center for AI (DFKI) more fundamental questions were investigated and the project was completed. Therefore, this report is also separated into two parts. In the first part a general model for constraints over hierarchies and the corresponding processing methods are explained. The second part gives some insights into the specific issues of the configuration task, especially when modeled as a constraint satisfaction problem. Thus, readers who are only interested in these problems may skip part one.

Unfortunately, since I changed my areas of research, this project will not be directly proceeded.

Kaiserslautern, July 1991

Contents

List of Figures

Chapter 1

Introduction

Constraints, naively being defined as consisting of a set of variables and a relation on them, play a crucial part in many fields of AI. Ever since Waltz [Wal72] proposed a methodology to recognize objects on the basis of their contours, the theoretical aspects and the practical issues of constraints have been intensively investigated.

Constraints are composed to build more complex structures by coupling single constraints via common variables. The resulting network is called a *constraint network*. The problem which must be solved is to find an assignment of values to the variables such that all constraints—respectively, the underlying relations—are satisfied simultaneously. This problem is called the *constraint satisfaction problem* (CSP). Depending on the requirements one needs to find either a single solution or all solutions for the CSP, defined as a constraint network.

The complexity of this class of problems is of great interest. Since the "four color problem"—a well-known $\mathcal{NP}$–complete problem—can be modeled as a constraint satisfaction problem, the problems in this class are $\mathcal{NP}$–complete as well. Consequently, the algorithms to solve the CSP have an exponential complexity in general, unless $\mathcal{NP} = \mathcal{P}$. Nevertheless, there are subclasses which can be solved in polynomial time, cf [Fre82].

The obvious method for solving the CSP over finite domains is the generate–and–test algorithm: All possible combinations of values for the variables are systematically generated and then tested whether they satisfy all constraints. An improvement of this approach is the chronological backtracking algorithm, which generates the assignment of values incrementally over the variables and checks whether a constraint is violated at each generation step. If a value violates a relation this value is retracted and the next one in the domain is tried. This approach still shows a "pathological" behavior [Mac77].

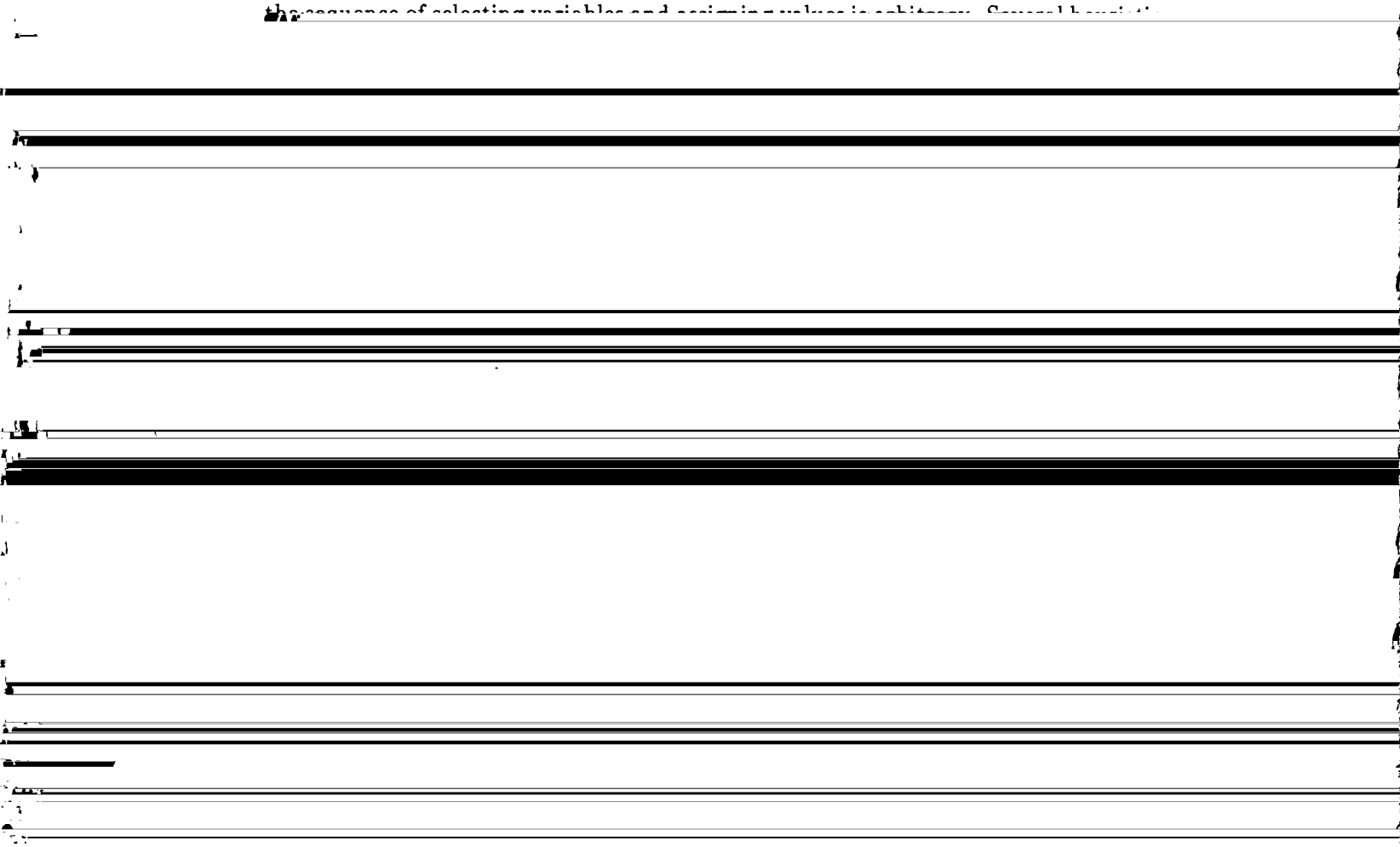
A key issue in research is how to cope with this pathological behavior. Various approaches can be found in literature:

- In a *preprocessing* step some values which cannot contribute to any solution are deleted from the domain of the variables. Basically, these algorithms have been developed from the Waltz labeling algorithm [Wal72] which transforms the network to a locally consistent state; hence, all remaining values of the variables contribute to at least one solution of the relation. All constraints are taken into account individually, thus no statement can be made about the contributions of the remaining values to the overall solution. Local consistency is discussed in more detail in section 2.3.1.

  Enhancements of this algorithm are those which take into account more than one constraint simultaneously or synthesize new, already implicitly defined constraints and add

them to the network, cf [Fre78]. They reach a higher level of consistency in the network, e.g. path consistency [Mac77], k–consistency [Fre82].

- The constraint network is transformed to a *relaxed network* with equal solutions but fewer backtracking steps (ultimately, without any backtracking step, cf [Fre82]) while the search for the global solution takes place. This is reached by exploiting the fact that the sequence of selecting variables and assigning values is arbitrary. Several heuristic



be used to choose a sequence which most likely implies restrictions on further assignments of values.

- Also *domain/task dependent heuristics* can be used to limit the absolute run–time. For example, a constraint is dynamically added to the constraint network at the time it is known that a special alternative or a set of alternatives imply an additional constraint [FBMB90], rather than starting with all possible constraints from the very beginning.

Based on these issues, various research topics concerned with constraints can be distinguished:

- An exhaustive research has been carried out developing efficient *constraint propagation algorithms* ([Mes89] and [Kum90]) working on CSPs over finite domains.
- Prolog, as being a very declarative programming language also based on relations, uses an uninformed backtracking algorithm to solve its CSP. Two limitations in the built–in Prolog control structure (resolution + backtracking) can be found and are tried to be overcome in a research field, called *constraint logic programming* (CLP).
  - The functional symbols in Prolog are uninterpreted and only used to syntactically structure the entities in the modeled world. The standard *unification* algorithm contains no information about terms which are semantically equal but syntactically unequal, e.g. set-of-elements(a,b,c) and set-of-elements(b,a,c). Since unification of

have to be maintained by the constraint handler. Hence, they do not have any operational or procedural meaning for the user the time the constraint network is defined. However, constraints have an attached operational aspect on a deeper level to perform their task for maintaining the relation. This level is hidden for the user and replaces the necessity to define these operations himself in conventional programming languages[1]. Their operational aspect, e.g. propagation and relaxation, is the main issue clarifying the relationship between constraints and relations[2].

Constraint-based modeling = Relations + Operations

The operations are specialized for different domains or, at least, exploit the structure of the constraints, cf [Ric89].

- Many problems in AI can be fact be regarded as search problems. The algorithms for solving the CSP cope with exponentially growing search spaces or infinite domains. The methods have been developed to prune the search space, thus, reducing the absolute run-time of the search algorithms. Therefore, constraint propagation mechanism are

  efficient solver for search problems

  in domains where searching is otherwise intractable.

- *Modeling complex domains and tasks* (configurations, diagnosis, action planning, etc.) is one of the key objectives of AI. Since constraints enable the knowledge engineer to focus on local relationships among entities in the domain, constraints and constraint networks are extremely suitable to model complex domains with a relatively simple and well defined formalism. Additionally, on account of the local behavior of the operations defined on constraints, some CSPs can be solved while using parallel algorithms [Kas89].

  Constraints describe local relationships between entities.

These characterizing properties were used to define a subclass of constraint satisfaction problems which are modeled in HieraCon: Dynamic constraint satisfaction problems with structured hierarchies as domains.

Motivation

Referring to the discussion above, the three main statements have been the guidelines for the theoretical aspects and the practical implementation of HieraCon.

1. Although taxonomies have already proved its declarativeness and its power by exploiting its internal structure for the inference process in logic programming [Mon87] very little research effort has been put into the development of constraints exploiting the structure of this kind of domains for its variables.

2. The problem definition and the problem solving process are strictly separated by providing a constraint language as a toolbox which can be used to define the knowledge cognitively adequate. The attached operations *propagation* and *relaxation* are defined for each type of constraint individually or, in general, by exploiting the structure of the domain. Therefore an object-oriented approach was chosen for the constraints where "propagation" and "relaxation" are attached methods for a whole class of constraints.

[1] Constraint languages may be regarded as programming languages for specific domains, cf [Lel88].
[2] From the same point of view Prolog can be regarded as 'horn-clauses + SLD-resolution'.

3. Various heuristics for the algorithms have been introduced in order to keep the problem solving process efficiently. Unfortunately, additional functionality like explanation and optimization features requires an overhead for data and dependency management. Consequently, the shortly mentioned algorithms for efficient constraint problem solving cannot be applied directly because they only exploit the finiteness of the domains but not its structure which is most important in the following application.

4. The user is able to attach constraints to individual entities of the knowledge base or to a group of entities via type declarations. So, the knowledge base can be built up incrementally and its maintenance is quite simple. The complexity appears only at run-time such that the problem solver has to cope with it and not the user.

5. Although the proposed general methodology to couple constraints and hierarchies was introduced from a general point of view a trial to apply it to a real AI task should reveal its feasibility.

6. Additionally, a proper design of the system on the conceptual level under consideration of the paradigms of the implementation language was an objective in order to improve and enhance the algorithms easily which process the constraints.

7. As conventional constraint propagation methods have already proven its power the presented method should be put into relation with them and include their functionality in specific cases.

However, due to time restriction only a part of the complete system could have been implemented. The implementation of a complete system must be an objective for further research.

Outline

As the following documentation is not a complete introduction to "constraints", readers who are not familiar with this research area are referred to the literature, basically cited in this introduction. Throughout the report, it was tried to discuss only the relevant topics exhaustively, whereas some hints can be found for literature presenting some issues in more detail.

The document is divided into two parts. Part I discusses the problem of integrating constraints with hierarchies of structured objects. Chapter 2 introduces the basic terminology in the field of constraint-based reasoning. A method for solving the constraint satisfaction problem over finite domains is presented. Subsequently, chapter 3 formally defines the object hierarchy based on an example of the kind of knowledge which has to be represented with the IS-A relation. In chapter 4 various kinds of constraints are introduced to represent a simple configuration task over the previously defined hierarchy. The last chapter in this part, chapter 5, describes how the various types of constraints are processed (constraint propagation and relaxation) by exploiting the structure of the domain.

In part II, based on a model for the configuration task in chapter 6, the design of the system—HieraCon—is discussed. The discussion is motivated both by exploiting the structure of the knowledge base as well as by the additional requirements evolving from the configuration task. Therefore, HieraCon is presented on a conceptual rather than on an implementation level. Finally, in chapter 9, a few other, already existing systems which influenced the design of HieraCon are discussed and put in a comparative relation with it.

Part I

Fundamentals

Chapter 2

Constraints

This chapter introduces a part of the basic terminology and the concepts used throughout the documentation. In the first two sections *constraints* and the *constraint satisfaction problem* (CSP) are defined. Subsequently, one approach to solve the CSP over finite domains—local propagation + backtracking—is discussed in detail. Readers who are familiar with these terms may skip to section 2.3.2 where the idea of coupling local propagation with hierarchical domains is introduced. Especially, this chapter may not be regarded as a complete introduction to algorithms for solving CSPs over finite domains. For that purpose, [Mes89] and [Kum90] provide a comprehensive overview.

2.1 Definitions and Concepts

Constraints are typically represented as relations over variables with attached domains; hence, the definition is based on the description of relations as subsets of Cartesian products.

Definition 2.1 *A finite set of variables $x_1, \ldots, x_n$ and their respective domains $D_1, \ldots, D_n$ is given. A n-ary* **constraint**[1] *$C(x_1, \ldots, x_n)$ over the variables $x_1, \ldots, x_n$ is a subset of the Cartesian product $D_1 \times \ldots \times D_n$.*

As the set of variables is finite a tuple—as instantiation of the Cartesian product—of discrete values is always suitable to represent a solution of a constraint.

Definition 2.2 *A* **constraint** *$C(x_1, \ldots, x_n)$* **is satisfied** *by a tuple $(d_1, \ldots, d_n)$ if $(d_1, \ldots, d_n)$ is in $C(x_1, \ldots, x_n)$.*

This definition does not state how the constraint is actually represented. It can be extensionally denoted by listing all the tuples which satisfy the constraint or it can be intensionally defined, describing it by a n-ary predicate which evaluates a given tuple. As one of the results of this work, it will be shown that even a mixture of both possibilities is sensible in case of a specific structure of the domain.

In general, constraints are in fact relational without distinguishable variables for the input or output of values with respect to the computation of the relation. Occasionally, constraints

[1] As in this definition, constraints may be regarded as relations over typed variables. Therefore, the terms "constraint" and "relation" will be used synonymously in case that only their representational character is

have a functional behavior on different levels. On the one hand, the user has a function in mind, e.g. an arithmetic function, while he defines the relationship between the variables. On the other hand, the constraints may be functionally computable the time they are evaluated. The following definitions identifies this subclass—*functional constraints*—in more detail.

Definition 2.3 *1. A constraint $C(x_1,\ldots,x_n)$ is* **functional** *with respect to variable x_k $(1 \leq k \leq n)$ if for all assignments of values to the variables x_j $(1 \leq j \leq n,\ j \neq k)$ there is exactly one value for the variable x_k such that $C(x_1,\ldots,x_n)$ is satisfied.*

2. A constraint $C(x_1,\ldots,x_n)$ is **completely functional** *if it is functional for all variables x_k $(1 \leq k \leq n)$.*

In this case, the solution tuples can be actively computed by the constraint itself and, therefore, it has a somehow different behavior than predicative constraints, being either extensionally or intensionally defined. Typically, arithmetic constraints are functional or even completely functional. The constraint $x+y=z$ with $D_x = D_y = D_z = \mathcal{Z}$ is completely functional, whereas $x * y = z$ is only functional with respect to z because the result of the division function may not be defined in $\mathcal{Z}$. Even in $\mathcal{R}$ it is not functional with respect to x or y because if 0 is assigned to y and z there are infinite many possible values for x. Consequently, one has to carefully select the domains and the properties of the functions which underly the constraint if one would like to work with functional constraints. This feature is very often used in case of infinite domains, such that the constraints are only evaluated if $n-1$ variables have assigned values as in [Güs89].

The benefit of functional constraints lies in their effective computation and their deterministic behavior. There is no need for generating tuples and testing them; instead, partial tuples can be enhanced to complete solution tuples. Additionally, as the value for some variables is uniquely determined this values must be used in all other constraints including the variable as well (cf definition 2.4). The disadvantage of functional constraints is that, though constraints are relations, the computation becomes directed and has to be deferred until all input variables of the function have assigned values.

Although the presented examples are defined over infinite domains the method of functionally computing values from other values can also be applied to finite domains

2.2 Constraint Satisfaction Problem

Single constraints can be coupled via common variables to build more complex structures.

Definition 2.4 *A finite set of variables $x_1,\ldots,x_n$ and their respective domains $D_1,\ldots,D_n$ is given. A* **constraint network** *is a set of constraints $C_1,\ldots,C_j$ where each constraint $C_k(x_{k_1},\ldots,x_{k_l})$ $(1 \leq k \leq j)$ is defined over the variables $x_{k_i} \in \{x_1,\ldots,x_n\}$ $(1 \leq i \leq l)$.*

Binary constraint networks are networks which consists only of binary and unary constraints. Two dual, graphical representations for constraint networks—labeled graphs—are used in the literature.

- Variables are represented as nodes and the constraints as links. This representation is basically utilized for binary constraint networks and will be used throughout this documentation, cf figure 2.2.
- In case that n–ary constraints are to be represented, very often the dual graph is chosen where nodes are representing constraints and links are representing their variables, cf [SJ80].

A constraint network is the complete graphical description of the problem which has to be solved.

Definition 2.5 *A* **constraint satisfaction problem (CSP)** *is a tuple $\langle \mathcal{V},\mathcal{C}\rangle$ where $\mathcal{V}$ denotes a finite set of variables with their corresponding domains and $\mathcal{C}$ denotes a set of constraints over these variables.*

The task is to find an assignment of values to all variables such that all constraints are satisfied simultaneously.

Definition 2.6 *A* **solution for the CSP** *$\langle \mathcal{V},\mathcal{C}\rangle$ with $\mathcal{V} = \{v_1,\ldots,v_n\}$, a set of variables with domains $D_1,\ldots,D_n$, and $\mathcal{C} = \{C_1,\ldots,C_k\}$, the set of constraints, is a tuple $(d_1,\ldots,d_n) \subseteq D_1 \times \ldots \times D_n$, such that for all $C_j \subseteq D_{j_1} \times \ldots \times D_{j_l}$ in $\mathcal{C}$, $(d_{j_1},\ldots,d_{j_l})$ with j_i fixed $(1 \leq j_i \leq n)$ is a solution for C_j.*

According to different requirements, one needs to find either one solution, all solutions, or the "best" solution for the given CSP.

2.3 Solving the CSP

The obvious approach for solving a CSP over finite domains is the *backtracking algorithm* described in figure 2.1 (adapted from [DP88]). *ComputeCandidates*$(x_1,\ldots,x_l,x_{l+1})$ returns the set C_{l+1} of possible values for x_{l+1} such that all constraints between x_{l+1} and $x_1,\ldots,x_l$ are satisfied by the current assignments for $x_1,\ldots,x_l$ and each value in the set C_{l+1}. The use of this algorithm reveals some maladies of backtracking which are discussed in [Mac77].

Forward$(x_1, \ldots, x_l)$
if $l = n$ then **exit** **Current Assignment**
$C_{l+1} :=$ *ComputeCandidates*$(x_1, \ldots, x_l, x_{l+1})$
if $C_{l+1} \neq \emptyset$
  then $d_{l+1} :=$ first element of C_{l+1}
    $x_{l+1} := d_{l+1}$
    remove d_{l+1} from C_{l+1}
    Forward$(x_1, \ldots, x_l, x_{l+1})$
  else *Go-back*$(x_1, \ldots, x_l)$
end **Forward**

Go–back$(x_1, \ldots, x_l)$
if $l = 0$ then **exit** **No Solution Exists!**
if $C_l \neq \emptyset$
  then $d_l :=$ first element of C_l
    $x_l := d_l$
    remove d_l from C_l
    Forward$(x_1, \ldots, x_l)$
  else *Go-Back* $(x_1, \ldots, x_{l-1})$
end **Go-Back**

Figure 2.1: Backtracking algorithm for solving the CSP

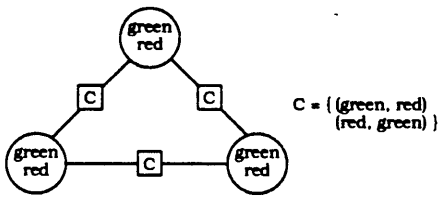


Figure 2.2: Graph coloring

2.3.1 Local Consistency

One of the inefficiencies is based on the fact that the sequence for the variables is fixed; hence, the following situation typically happens: There is a constraint $C(x_i, x_j)$ and the value $a \in D_i$ is assigned to x_i but there is no value $b \in D_j$ such that (a, b) satisfies $C(x_i, x_j)$. The backtracking algorithm checks all possible assignments $(a, d_{i+1}, \ldots, d_j)$ before it finally retracts a from the list of possible candidates for x_i. Obviously, this inefficiency can be prevented if one would check whether the domain for x_j includes a solution for $C(x_i, x_j)$ as soon as a is assigned to x_i. The following definition defines this state as *local consistency*.

Definition 2.7 *Constraint* $C(x_i, x_j)$ *is* **locally consistent** *if for each value* $d_i \in D_i$ *there is a value* $d_j \in D_j$ *such that* (d_i, d_j) *satisfies* $C(x_i, x_j)$.

Obviously, the definition states only the existence of such a value, but does not require that this value contributes to a solution for the constraint network at all. As the condition of local consistency is weaker then the one for a global solution, there might be a locally consistent state for a CSP but a solution for the CSP does not exist. Vice versa, if a global solution exists then the CSP also has a locally consistent state. A simple example can be found in figure 2.2. The constraint between two nodes is assumed to be '$\neq$'. The constraint network is obviously locally consistent: For each color in each node there is another color in the neighbored node such that the two colors are different. But a solution for the CSP does not exist: If two variables have the assigned values 'red' and 'green' there is no possible value left for the third variable.

The algorithm for achieving local consistency in a constraint network is straightforward and shown in figure 2.3. *LC* is called with a representation of the complete constraint network. The internal call *NC* (node consistency) evaluates all unary constraints, representing the initial value restriction for a variable. The algorithm runs until no more changes can be found in the constraint network. The efficiency can be improved because the above algorithm causes

```
LC(G)
for i := 1 until n do NC(x_i)
Q := {(i,j) | (i,j) ∈ arcs(G), i ≠ j}
repeat
        Change := false
        for each (i,j) ∈ Q do
            Change := (Revise(x_i, x_j) or
                       Change)
until Change = false
end *LC*
```

```
Revise(x_i, x_j)
Delete := false
for each d_{x_i} ∈ D_{x_i} do
    if there is no d_{y_i} ∈ D_{x_j} such that C(x_i, x_j)
      then remove d_x from D_{x_i}
           Delete := true
exit Delete
end *Revise*
```

Figure 2.3: Algorithm for achieving local consistency

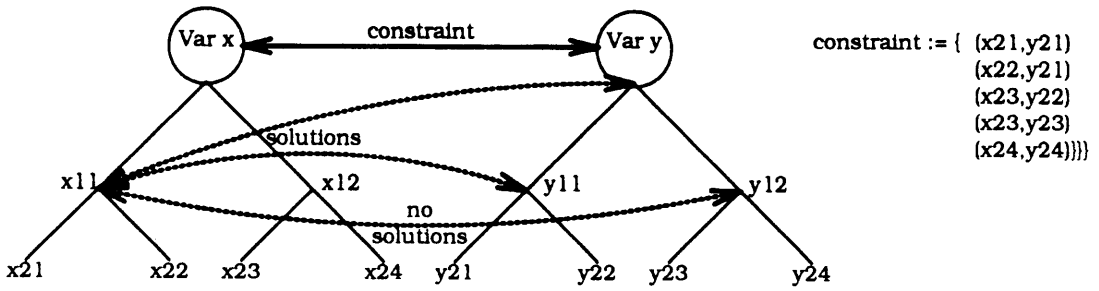


Figure 2.4: Local consistency over hierarchical domains

all arcs to be revised after a successful deletion of a value, though only a few are affected, cf [Mac77].

The usage of the algorithm as a preprocessing step for backtracking prevents the above discussed malady. Enhancements of this scheme are easily imaginable: Taking into account more than two variables in the preprocessing step would reach are higher level of consistency (cf [Fre78]); but these approaches are beyond the scope of this work.

2.3.2 Exploiting Hierarchies for Local Consistency

The *LC* algorithm in figure 2.3 for achieving local consistency starts with an initialization of all values to the variables in the network. Very often, the domains are structured as hierarchies where the "IS–A" relationship represents the subset relation between two concepts. Chapter 3 describes the terminology *hierarchy* in more detail. In that case, it is preferable to assign a concept to a variable instead of all the individuals explicitly for which the vertex is an abbreviation. In [MMH85] the *LC* algorithm is enhanced for hierarchical domains.

The idea may be summarized as follows. The solution tuples of the constraints are defined as in case of flat domains. The constraints are evaluated in a preprocessing step such that a predicate is attached to each concept in the hierarchy of the domain. This predicate stores the relation between two concepts: whether there are solutions of the constraints or not. The

domains in the negative case without investigating each individual and checking the constraint several times. For example, if the current value of x is $x11$ and the current value of y is $y12$ the information which is stored in concept $x11$ states that there is no solution for the CSP at all. Consequently, this technique saves calls of the predicate $C(x, y)$ in the conventional version of LC in figure 2.3. The improvement of the complexity is also investigated in [MMH85].

In the above approach, the constraints must still be extensionally denoted on the individuals' level. An obvious enhancement is to define the constraint solutions already on the concept level. This idea has been worked out in [MJ91] and will be used in HieraCon as well.

From the viewpoint of constraint processing the idea of functional constraints can be found here again. Describing the extensional solution as relationship between concepts enables a functional propagation on a higher level: As soon as one of the concepts is assigned to a variable the other variable gets the other concepts as value which hold the solutions of the local relationship. This idea is also adopted in HieraCon where the constraints are defined as relations between classes in the knowledge base.

Chapter 3

Structured Hierarchy

This chapter defines the structure of the domain of the CSPs taken into account. A general review of the properties of hierarchies under special consideration of the IS–A relation is given in the first section. Subsequently, an example of a real domain is presented to motivate the required expressive power of the representation formalism. Finally, a formal definition of the IS–A relation is evolved to gain a semantics on which the constraints will be based in the next chapter.

3.1 Concepts

The observation that an extensive part of the knowledge AI has to cope with can be represented with the aid of conceptual relations between objects has led to the development and widespread use of hierarchies. The conceptual relations in common use are either the IS–A relation, which describes a generalization/specialization relationship between objects or the PART–OF relation, denoting a composing/decomposing relationship. The decision which one to choose depends on the intention of what should be represented and how the hierarchy is used for knowledge processing. As constraint solving is a process of refining values for a set of variables[1] and the solution is the composition of the single values in the constraint network, the hierarchy is necessarily being built over the IS–A relation, whereas the single elements of the solution are decomposed and represented in the layout of the network. The IS–A relationship then structures the domains of the variables and is being used for refining the value of a variable from its initial value—a class—down to a leaf of the hierarchy—an individual.

3.1.1 Hierarchies

The conceptual description of the hierarchy can be performed in two different ways.

- Terminological representation languages, e.g. KL–ONE [BS85] or Krypton [BFL83], provide subsumption algorithms which compute the hierarchy automatically by the use of the conceptual description of the objects itself. Consequently, the languages must be based on a formal semantics such that the IS–A relation is well–defined and correctly computable[2]. Originally, terminological representation languages were developed for natural language understanding, but may also have some impact on representing technical knowledge [BH91] in expert systems.

[1] [Fox86] designates *constraint solving*, besides *search* and *reasoning*, as one of the major point of views for problem solving tasks.

[2] However, subsumption in KL–ONE is undecidable! [SS89a]

- The hierarchy may be defined by any construct in the representation language that represents "IS–A". In this case, there need not to be a semantical relationship between the objects which were defined as being in the IS–A relation with each other. Additionally, the user is able to define inconsistent knowledge bases with consistent objects. This method is usually provided by object oriented languages with all its problems based on an informal and inconsistent semantics, e.g. multiple inheritance, overwriting of default values, etc.

The approach taken here is located anywhere in between. The hierarchy is defined just by using a construct for the IS–A relationship but vice versa if two objects are comparable with each other, they must fulfill some formal criteria. This is also required to get a well defined propagation algorithm which partially works on the structure of the hierarchy and assumes that it is semantically conform. Additionally, multiple inheritance also gets a well defined semantics and default values can be overwritten if it happens in consistence with a type hierarchy.

In the following, the terms **class** and **individuals** are used for the objects of the hierarchy. An *individual* represents a physical object in the world and is a leaf in the hierarchy. A *class* represents a set of individuals with common properties and thus, may keep properties which must hold for each of them. Classes are the inner nodes of the hierarchy.

3.1.2 The IS–A Relation

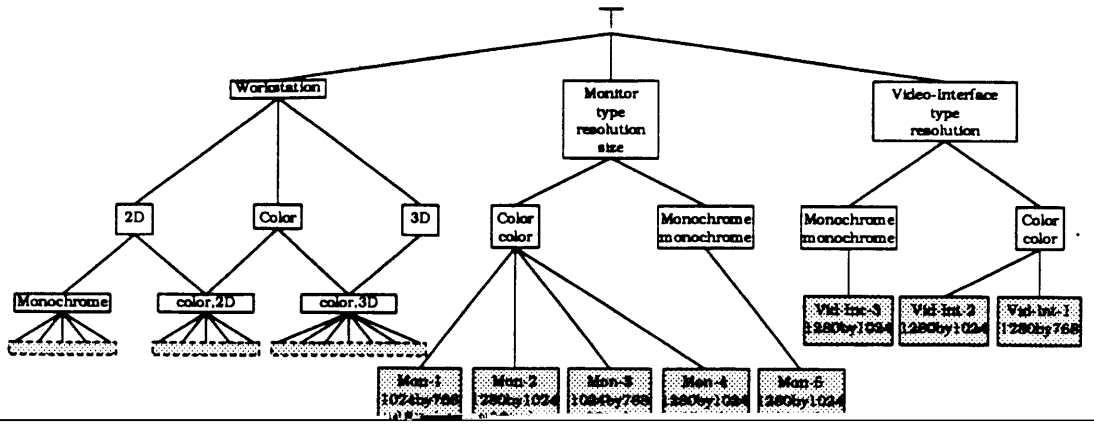
The hierarchy is composed by means of the IS–A relation between the objects of the hierarchy. There are many different possible interpretations for the IS–A relation [Bra83]. In order to clarify its semantics the two relations—between two classes and between a class and an individual—must be explained. The used example will be comprehensively explained in section 3.2 but should be intuitively clear.

- The *relation between two classes* is regarded as an abstraction/specialization relationship in the following way: If `Monitor`(v) is a predicate (or an unary constraint) taken to be an abstraction of `Color-Monitor`(v) then the IS–A relationship between them is interpreted as "for every individual v if `Color-Monitor`(v) then `Monitor`(v)". From a constraint reasoning point of view this relationship can be regarded as a domain restriction by an unary constraint, e.g. `Color-Monitor` is the set of all individuals of the domain `Monitor` for which the unary constraint =(`type`(x), `color`) is satisfied.
- The *relation between an individual and a class* is handled as a predicate (or unary constraint) where IS–A is applied to an individual involving a type predicate, e.g. if `Monitor-1` is the individual and `Color-Monitor` the class, IS–A expresses the fact that `Color-Monitor(Monitor-1)`. Consequently, *classes* are symbolic descriptions of sets of all *individuals* for which the corresponding type predicate holds.

This informal description of the point of view for the IS–A relation will be formally defined in section 3.3.

3.1.3 Structured Objects

It already follows from the above example that the objects in the hierarchy are not flat, only denoted by their class specifier. Additionally, the objects have an internal structure in the

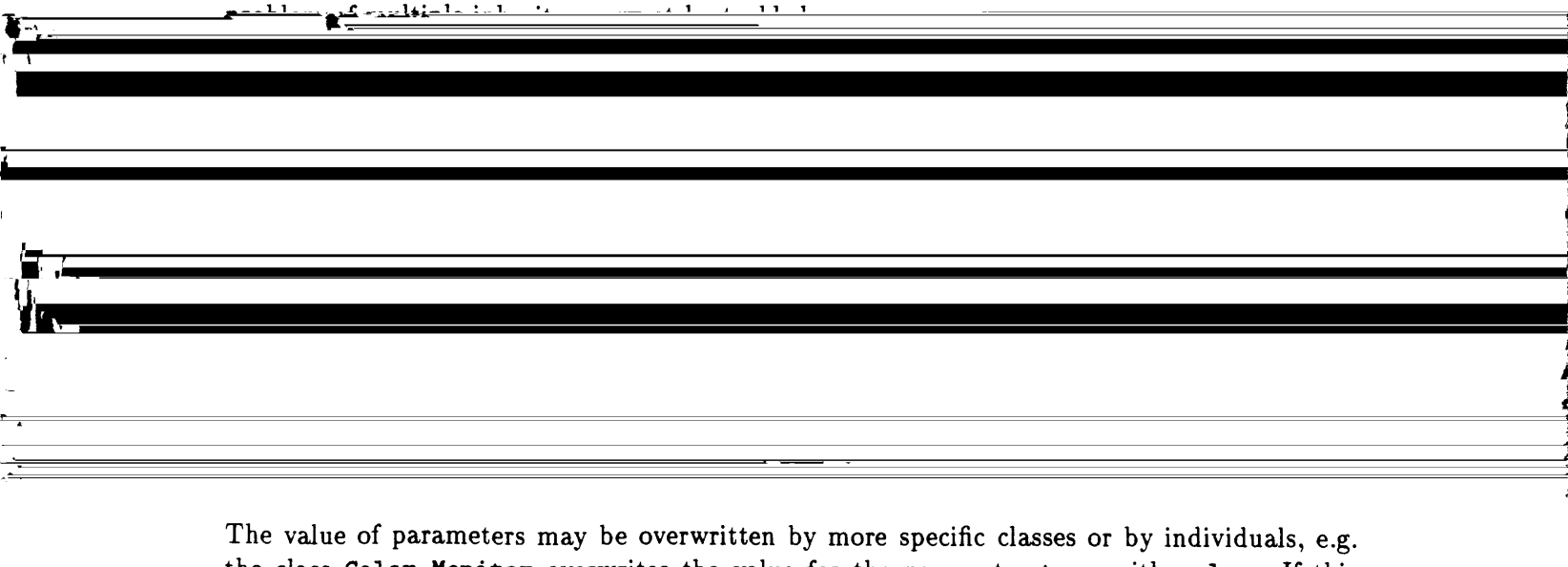


sense of frames: Classes and individuals are concretized with attribute/value pairs. In fact the attribute/value pairs could be factored out by introducing a new class for each value with the attribute as relation between the object and the new class. This approach is used in semantic networks. Nevertheless, defining physical objects in technical domains by their attributes in a frame–like data structure at least minimizes the size of the hierarchy, but is also much more declarative. Hierarchies are also used to inherit values for attributes along the IS–A relationship.

The term **structured hierarchy** will be used in the following to denote a hierarchy over the IS–A relation as defined above whose objects—classes and individuals—have an additional internal structure.

3.2 An Example

parameters. This part of the hierarchy also has a lattice—like structure. Consequently, the



The value of parameters may be overwritten by more specific classes or by individuals, e.g. the class `Color-Monitor` overwrites the value for the parameter `type` with `color`. If this specification is well-defined there must a relationship between `Monitor-Type` (which is a type) and `color` (which is a constant).

Individuals are characterized by attaching constants to all of its parameters, which is cognitively adequate because they represent the physical objects of the modeled world which are uniquely identified by theses constants.

3.3 Definition

In order to gain a well–defined hierarchy which is capable of representing the knowledge of the kind of the above example, the hierarchy is formally defined.

The language for the knowledge base consists of the partially ordered, finite sets of symbols for the

primitive types $\mathcal{PT} := (\{\top, \bot, Pt_1, Pt_2, \ldots\}; isa)$
structured types $\mathcal{ST} := (\{\top, \bot, St_1, St_2, \ldots\}; isa)$

—with 'isa' as the transitive, reflexive, and antisymmetric ordering relation—and the following sets of functions with its interpretations for the

parameters $\mathcal{P} := \{p_1 : St_i \rightsquigarrow Pt_j, p_2 : St_k \rightsquigarrow Pt_l, \ldots\}$
constants $\mathcal{C} := \{c_{i_1} : Pt_i, c_{i_2} : Pt_i, \ldots\}$
individuals $\mathcal{IND} := \{Ind_{i_1} : ST_i, Ind_{i_2} : St_i, \ldots\}$

'$\rightsquigarrow$' denotes that this is a partial function. Parameters with the same name but different types are regarded as being different. All sets are disjunctive from each other except for $\mathcal{PT} \cap \mathcal{ST} = \{\top, \bot\}$ with $\bot\ isa\ T\ isa\ \top$ for all types $T \in \mathcal{PT} \cup \mathcal{ST}$. Consequently, $\mathcal{PT} \cup \mathcal{ST}$ forms a semi–lattice with $\top$ (top) as upper bound and $\bot$ (bottom) as lower bound.

Note that the interpretations for the constants ($c : PT$) and the individuals ($ind : ST$) correspond to the interpretation of the IS–A relation as type predicates: $PT(c)$ and $ST(ind)$, cf section 3.1.2.

The sets for the above example are then defined as follows:

$\mathcal{PT}$:= ({$\top, \bot$, `Monitor-Type`, `IntegerByInteger`, `Integer`}; isa)
$\mathcal{ST}$:= ({$\top, \bot$, `Workstation`, `2D-workstation`, `Color-Workstation`, `3D-Workstation`,

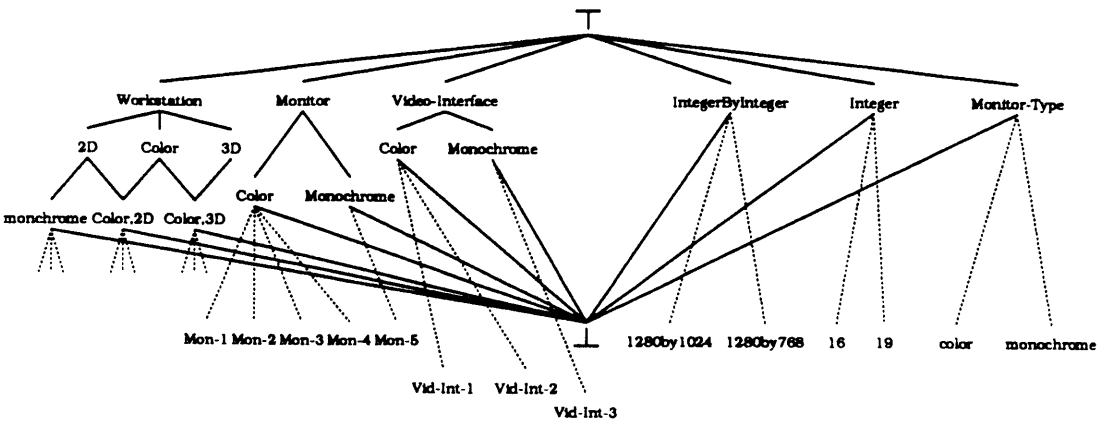


Figure 3.2: A hierarchy of structured and primitive types

$\mathcal{C}$:= { color: Monitor-Type, monochrome: Monitor-Type,
1024by768: IntegerByInteger, 1280by1024: IntegerByInteger,
16: Integer, 19: Integer}
$\mathcal{IND}$:= { Mon-1: Color-Monitor, Mon-2: Color-Monitor, Mon-3: Color-Monitor,
Mon-4: Color-Monitor, Mon-5: Monochrome-Monitor,
Vid-Int-1: Color-Video-Interface, Vid-Int-2: Color-Video-Interface,
Vid-Int-3: Monochrome-Video-Interface}

The isa relation over $\mathcal{PT}$ and $\mathcal{ST}$ is shown in figure 3.2 and the constants and individuals are added by the IS–A relation (in dotted lines).

In order to gain the interpretation of the IS–A relation between individuals and its classes and an ordering of all objects in the knowledge base Λ the *isa* relation over $\mathcal{PT}$ and $\mathcal{ST}$ is enhanced.

Definition 3.1 *1. c isa PT if c: PT' ∈ C and PT' isa PT*

2. Ind isa ST if Ind: ST' ∈ C and ST' isa ST

Eventually, the hierarchy in figure 3.2 is defined where all links become the *isa* relation as reflexive, transitive, antisymmetric ordering relation on this hierarchy.

The objective is to build a knowledge base which basically represents the knowledge in figure 3.1 by using the above sets and a composition rule for the structured objects.

Definition 3.2 *The* **knowledge base** Λ *consists of*

1. *a set of structured types* $\mathtt{St}\begin{bmatrix} \mathtt{p_1} & \mathtt{Pt/c_1} \\ \vdots & \vdots \\ \mathtt{pn} & \mathtt{Pt/c_2} \end{bmatrix}$ *with* $\mathtt{St} \in \mathcal{ST}$, $\mathtt{Pt/c_i} \in \mathcal{PT} \cup \mathcal{C}$, *and* $\mathtt{p_i} \in \mathcal{P}$ *with* $p_i: St \rightsquigarrow Pt_i$

2. *a set of individuals* $\mathtt{Ind}\begin{bmatrix} \mathtt{p_1} & \mathtt{c_1} \\ \vdots & \vdots \\ \mathtt{pn} & \mathtt{cn} \end{bmatrix}$ *with* $\mathtt{Ind} \in \mathcal{IND}$, $\mathtt{c_i} \in \mathcal{C} \cup \bot$, *and* $\mathtt{p_i} \in \mathcal{P}$ *with* $p_i: St \rightsquigarrow Pt_i$ *and* $c_i: Pt_i \in \mathcal{C}$

3. *the set $\mathcal{PT}$ of primitive types*

4. *the set $\mathcal{C}$ of constants*

The notation for the structured types and individuals was already used in section 3.2 where some examples can be found. $\perp$ is introduced as a possible parameter value to allow the representation of objects with unspecified characteristics.

The isa relation is now defined for all objects which play a role in the knowledge base. Consequently, the IS–A relation as informally described in section 3.1.2 is now defined by use of the

$$\mathtt{St}_2 \begin{bmatrix} \mathtt{p}_1 & \mathtt{Pt/c}_{2_1} \\ \vdots & \vdots \\ \mathtt{p}_n & \mathtt{Pt/c}_{2_n} \end{bmatrix} \textit{ IS–A } \mathtt{St}_1 \begin{bmatrix} \mathtt{p}_1 & \mathtt{Pt/c}_{1_1} \\ \vdots & \vdots \\ \mathtt{p}_n & \mathtt{Pt/c}_{1_n} \end{bmatrix}$$

with $St_1, St_2 \in \mathcal{ST}$, $p_i \in \mathcal{P}$, *and* $Pt/c_{1_i}, Pt/c_{2_i} \in \mathcal{PT} \cup \mathcal{C}$

$$\textit{if } St_2 \textit{ isa } St_1 \textit{ and } \forall p_i\,(1 \leq i \leq n)\; Pt/c_{2_i} \textit{ isa } Pt/c_{1_i}$$

and for all structured types $\mathtt{St} \begin{bmatrix} \mathtt{p}_1 & \mathtt{Pt/c}_1 \\ \vdots & \vdots \\ \mathtt{p}_n & \mathtt{Pt/c}_n \end{bmatrix}$

$$\perp \textit{ IS–A } \mathtt{St} \begin{bmatrix} \mathtt{p}_1 & \mathtt{Pt/c}_1 \\ \vdots & \vdots \\ \mathtt{p}_n & \mathtt{Pt/c}_n \end{bmatrix}$$

hierarchy. This is required because otherwise constraints could not exclude complete classes due to the value of some individual parameters because the value satisfying the constraint could be defined in one of the subclasses. Multiple inheritance for parameters (primitive types or constants from various supertypes) is allowed in case that the inheritance is compatible with respect to the ordering relation over primitive types. Consequently, a class does not need to hold a supertype preference list defining the sequence of inheriting values from the supertypes.

Finally, the knowledge base over which the constraints will be declared is defined.

Definition 3.5 *The* **knowledge base** Λ *is a six-tuple* $\langle \mathcal{PT}, \mathcal{ST}, \mathcal{IND}, \mathcal{C}, \mathcal{P}, IS\text{-}A \rangle$ *with the sets as described above and IS-A as defined in definitions 3.3 and 3.4*

Summarized it may be said that with definitions 3.3 and 3.4 the IS–A relationship as discussed

in section 3.1.2 is put on a formal basis such that the value propagation of constraints exploiting the structured hierarchies can be defined.

Chapter 4

Constraint Typology

In this chapter the various types of constraints required to model a simple configuration task are introduced. In the first section, based on an elementary example, the objects of the hierarchy are investigated from a constraint reasoning point of view and the kinds of occurring constraints are discussed. The requirements for representing the CSP (cf definition 2.5)—constraints and variables—are derived. Finally, the constraint typology for $H^{iera}C_{on}$ is introduced and its semantics and representational character is investigated. The procedural aspect of these kinds of constraints are discussed in chapter 5.

4.1 Example of a CSP

The intention is to use the CSP under consideration for a configuration task, circumstantially discussed in chapter 6. A motivating example is presented in this section from which the kind of required constraints can be derived.

Focussed on local consistency (cf definition 2.7), the example includes only two variables but is suitable to reveal the basic principles and requirements. Referring back to figure 3.1, the CSP is assumed to consist of the variables *monitor* and *video-interface* and the sets of individuals with the same name (`Monitor` and `Video-Interface`) as domains. Informally, the constraint between these variables must express that only two compatible individuals—a monitor and a video-interface—can be connected with each other. Compatibility in this example can be described on the level of the parameters (`monitor-type` and `resolution`) which must have equal values. Figure 4.1 shows the CSP together with the domains and the solution of the constraint between the variables. The solution of the constraint is defined conventionally: All solution tuples are listed explicitly; thus, the predicate in the algorithm for deriving local consistency works with this set of solution tuples. The variables are assumed to be flat in the sense that they can only hold an individual by its name but cannot represent the internal structure of the objects as well.

In the following, the kind of knowledge which is used for representing both, the structured hierarchy and the constraint solutions, is investigated from a constraint reasoning point of

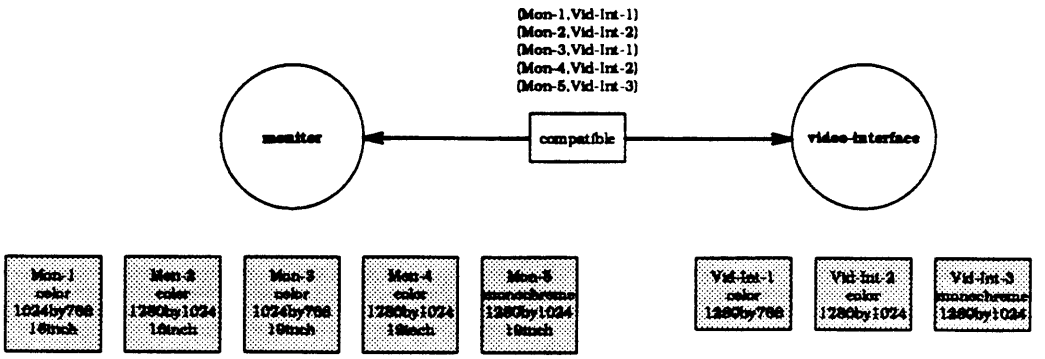


Figure 4.1: Example of a CSP

4.1.1 Extensional Denotation

All possible value pairs of the relation between **Monitor** and **Video-Interface** may be extensionally denoted as {(**Mon-1**, **Vid-Int-1**), (**Mon-2**, **Vid-Int-2**), (**Mon-3**, **Vid-Int-1**), (**Mon-4**, **Vid-Int-2**), (**Mon-5**, **Vid-Int-3**)}. With this representation classical constraint propagation algorithms can be applied. For example, in order to keep the relation locally consistent (definition 2.7) specific values for the variable *Video-Interface* directly implies a restriction of possible assignments for the variable representing the class **Monitor**.

The disadvantage of this approach is that there is no way to refer to individuals via the value of a parameter, e.g. if all color monitors are required each individual of the class **Monitor** must be selected and the parameter **type** is investigated whether it has the specific value **color** or not; theoretically, if the parameters cannot accessed even a new constraint must be defined whose solution are the color monitors **Mon-1**, ..., **Mon-4**. In addition, complete individuals are assigned to a variable, although later on some of them may be retracted by other constraints requiring for specific values in parameters.

Consequently, two requirements are imposed:

1. As far as classes are already defined in the knowledge base, they should be used as an abbreviation for complete sets of individuals, adapting the point of view that classes are symbolic descriptions of such sets.

2. Parameters should be accessible in order to define an unary constraint over them. This constraint then represents a restriction of the initial domain and works as a filter for the set of individuals.

A mixture of both approaches follows from the fact that some classes are just defined by factoring out values for parameters, e.g. the class **Color-Monitor** includes all individuals whose value for the parameter **type** is **color** or, expressed as an unary constraint, **Monitor**(x)$\wedge$ =(**type**(x), **color**).

4.1.2 Individuals as n–ary Constraints

The structured individuals of the knowledge base can themselves be represented as relations. Here, the parameters are regarded as binary constraints between the individual specifier and the parameter's value where the name of the parameter is the name of the relation. For example,

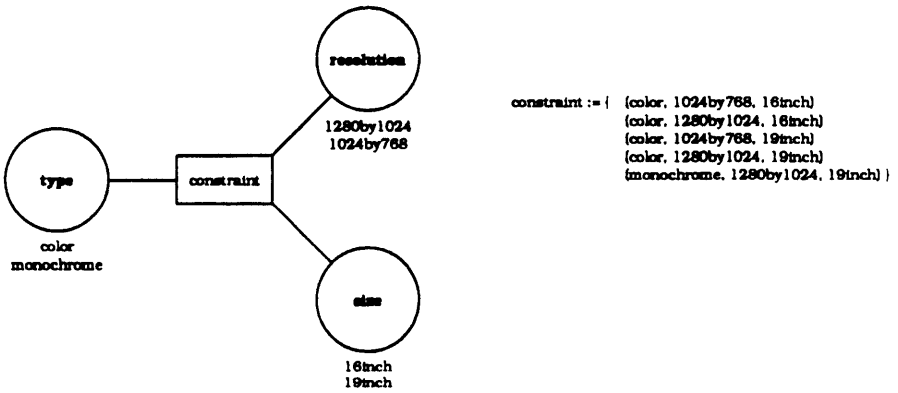


Figure 4.2: Representing individuals as n–ary constraints

the above individual **Mon-1** can be represented with three binary relations: **type(Mon-1, color)**, **resolution(Mon-1, 1024by768)**, and **type(Mon-1, 16inch)**.

The disadvantage of this approach is that new variables for each parameter must be introduced and that the explicit information—the three values **color**, **1024by768**, and **16inch** occur together in one individual—is factored out and then stored implicitly in one argument of the relation. A more declarative representation with n–ary constraints is shown in figure 4.2.

A parameter must be interpreted via the fixed position in the relation describing an individual. Thus, the first monitor is represented as the triple (**color**, **1024by768**, **16inch**). The first argument position is interpreted as representing the value of the parameter **type**, the second as the value of the parameter **resolution**, etc. Because more combinatorical combinations can be built which are not present in the domain, e.g. (**monochrome**, **1024by768**, **19inch**), the relation must be explicitly restricted to the five individuals by denoting the triples of the relation as described above.

The disadvantage of this approach is obvious. Although the relationship between specific values of the parameters are already represented in the knowledge base, they are factored out by flattening the individuals in figure 4.2. Consequently, they must be reintroduced via a

suitable representation of the relation describing the individuals. This problem is also discussed in [MF87]. Additionally, it is rather difficult to attach other characteristics to the individuals, e.g. constraints, except for introducing the specifier, e.g. **Monitor-1**, via an additional variable.

As a result for constraint processing, one could state that it is preferable to enhance the algorithms for constraint handling in order to cope with structured objects rather than compiling these objects into a conventional CSP with flat domains.

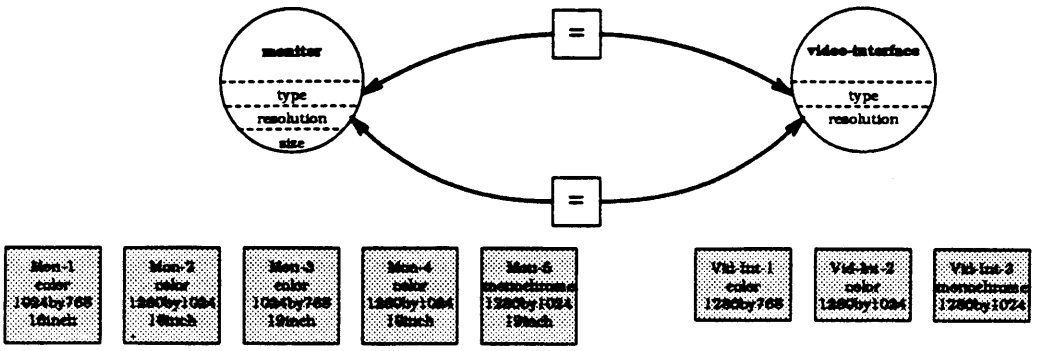


Figure 4.3: Constraint as relations between parameters

figure 2.3) is running. Both algorithms could be improved by exploiting the structured hierarchy in the following way: If a specific value for a parameter is required and a class exists which represents this value, the algorithm computes the intersection of the current assignment for the variables with all individuals satisfying the constraint. The intersection is computed by determining the greatest common subclass (definition 5.1) of the two classes. If such a class does not exist then the constraint propagation algorithm must still work on the parameter values. Consequently, the conventional approach is included in the above scheme.

Summarized it may be said, that the internal structure of the domain should be exploited as best as it can in order to reduce the number of predicative tests by a relatively simple computation on the level of classes. Additionally, the unsatisfiability of a CSP may be detected without investigating the properties of the single individuals.

4.2 Requirements for the CSP

From the above discussion and the specific properties of the domain the requirements for defining a CSP over structured domains are as follows:

- The variables of the CSP are structured in order to represent the internal structure of the objects in the knowledge base. The constraint solver must be capable to access to these values.
- In addition to constraints, restricting the values of parameters, a new type of constraints must be introduced which explicitly exploits the hierarchical knowledge. Solutions of constraints are then described by attaching these constraints to the classes.
- The conventional approach of denoting constraint—listing of the solution tuples—should be included by the constraint model, e.g. the backtracking algorithm should be supported by reducing its absolute run-time but should work as known as if the constraints were conventionally defined.

In the next section, a typology for the constraints will be explained in detail. Their operational behavior is discussed in chapter 5.

4.3 Constraint Typology

In this section, the various types of constraints required to model a CSP over structured domains are defined. The point of view is adopted that constraints exist between variables and

the solution of the constraint may be described by various terminologies or even a mixture of them. In figure 4.1 an extensional denotation is used whereas figure 4.3 gives an example where the same constraint is described by two '=' relations between parameters. Both notations designate the same solutions and may be regarded as being equal from this point of view. However, the second approach exploits much better the information which was already decoded in the hierarchy and enables an easier maintenance of the knowledge base: New individuals can be added in their "right" place without changing the constraint because it must also be fulfilled on the parameter level by the new individuals.

The following terminology for describing solutions of constraints is defined:

- **class–rest**(x: St) (class restriction) is an unary constraint expressing that the class of a variable x must be specialized to the class St which must be a subclass of the variable's class.
- **not–class–rest**(x : St) (not-class restriction) specifies that variable x must not hold class ST or one of its subclasses.
- **class–comp**(x_1 : St_1, x_2 : St_2) (classes compatible) is a binary constraint between x_1 and x_2 which states that two subclasses of the variables are compatible with each other. Specifically, if the current class St' of variable x_1 IS–A St_1, then the current class St'' of x_2 must be subclass of St_2 and vice versa.
- **class–incomp**(x_1 : St_1, x_2 : St_2) (classes incompatible) is a binary constraint between x_1 and x_2 declaring that a class for one variable is incompatible with a class for another variable, i.e. if the current class St' of variable x_1 IS–A St_1, then the current class St'' of x_2 must not be a subclass of St_2 and vice versa.
- **param–rest**(**unary–pred**, $p(x)$) (parameter restriction) states that all individuals or classes of the variable must fulfill the predicate "unary–pred" on one of its parameters p.
- **param–comp**(**pred**, $p_1(x_1), p_2(x_2), \ldots, p_n(x_n)$) (parameter compatibility) is a predicate between the variables $x_1, x_2, \ldots, x_n$ which fixes a predicate "pred" between the parameters of the classes. The previously defined constraint *param–rest* is a special case of this one but was introduced for compatibility reasons. In contrary to the constraints over classes, the negation of the constraint was not introduced to the typology but has to be coded in the predicate itself. For example, the negation for the the constraint **param–comp**(=, `resolution`(*monitor*),`resolution`(*video–interface*)) must be defined as **param–comp**($\neq$, `resolution`(*monitor*),`resolution`(*video–interface*))

The *class–comp* and *class–incomp* constraints may also have individuals as parameters. In this case the extensional solution of a constraint is denoted. The unary constraints may be regarded as a specialization of the binary constraints where one argument holds a fixed value. They are used to make an instance of the CSP by restricting possible values for one or more variables. These restrictions may be user requirements, heuristics, external data, etc.

Referring back to the CSP in figure 4.1, the constraint between the variables *monitor* and *video–interface* can be defined in various ways which are all equal from the point of the solutions for the constraint.

1. The extensional denotation on the individuals' level without exploiting the hierarchy is denoted as
   class–comp(*monitor*: `Mon-1`, *video-interface*: `Vid-Int-1`)

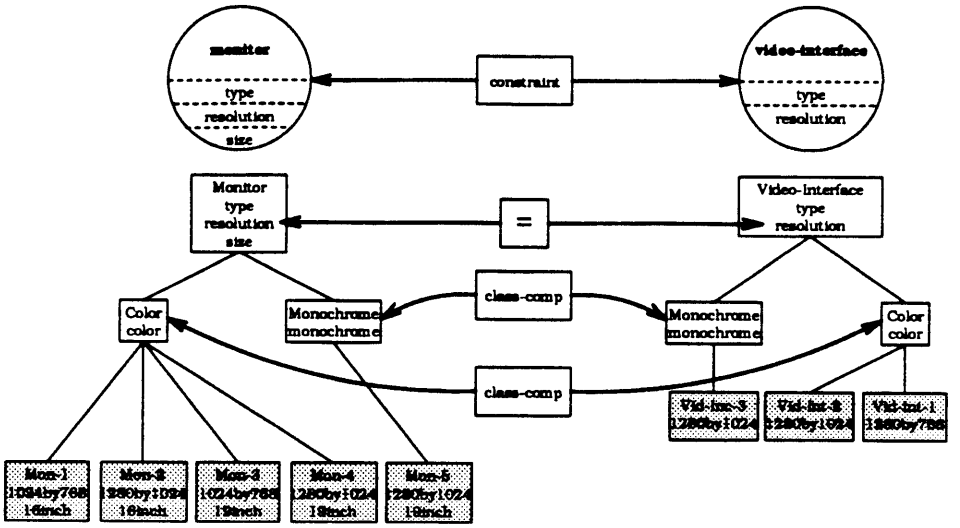


Figure 4.4: Constraints which exploits the structured hierarchy

class–comp(*monitor*: `Mon-2`, *video-interface*: `Vid-Int-2`)
class–comp(*monitor*: `Mon-3`, *video-interface*: `Vid-Int-1`)
class–comp(*monitor*: `Mon-4`, *video-interface*: `Vid-Int-2`)
class–comp(*monitor*: `Mon-5`, *video-interface*: `Vid-Int-3`)
and corresponds exactly to the notation of solution tuples.

2. The constraint may also be denoted as binary = relation between the parameters `type` and `resolution`.
param–comp(=, `type`(*monitor*), `type`(*video-interface*))
param–comp(=, `resolution`(*monitor*), `resolution`(*video-interface*))
This corresponds exactly to the CSP as defined in figure 4.3 and only exploits that the variables are structured but not the fact that the IS–A hierarchy itself contains useful information.

3. A mixture of two types of constraints under consideration of the structured hierarchy is defined:
class–comp(*monitor*: `Color-Monitor`, *video-interface*: `Color-Video-Interface`)
class–comp(*monitor*: `Monochrome-Monitor`,
video-interface: `Monochrome-Video-Interface`)
param–comp(=, `resolution`(*monitor*), `resolution`(*video-interface*))
This notation is graphically represented in figure 4.4.

   The notation already reveals the idea that the constraints are not added to the constraint network itself but are assigned to classes in the hierarchy. As soon as this class is assigned to a variable, the corresponding constraint is activated and added to the network dynamically. Consequently, in the above case would the constraint between the parameters `resolution` would be statically added to the network because the classes the constraint is attached to, `Monitor` and `Video-Interface`, are assigned to the variables as initial value. Contrarily, the constraints between the classes will be added as soon as this class is assigned to one of the variables or one of the individuals of the class is selected.

So far, some ideas of how the constraints are processed have already been mentioned. In the next chapter, these approaches are discussed in more detail and the algorithms are presented.

4.4 CSP over the Knowledge Base

With the notation of chapter 3 the CSP with the above typology and the knowledge base Λ can be defined.

Definition 4.1 *A* **CSP** *over* $\Lambda := \langle \mathcal{PT}, \mathcal{ST}, \mathcal{IND}, \mathcal{C}, \mathcal{P}, \mathit{IS\text{-}A} \rangle$ *is a tuple* $\langle \mathcal{V}, \mathcal{CS} \rangle$ *with*

- $\mathcal{V}$ *is a finite set of typed variables* $\{x : Cl, x_1 : Cl_1, \ldots, x_n : Cl_n\}$
  with Cl_i *is a structured class in* Λ
- $\mathcal{CS}$ *is a set of constraints* $\{C_1, C_2, \ldots\}$ *where* C_i *has one of the following forms:*
  - *class-rest*$(x : St)$
  - *not-class-rest*$(x : St)$
  - *class-comp*$(x_1 : St_1, x_2 : St_2)$
  - *class-incomp*$(x_1 : St_1, x_2 : St_2)$
  - *param-rest*(*unary-pred*, $p(x)$)
  - *param-comp*(*pred*, $p_1(x_1), p_2(x_2), \ldots, p_n(x_n)$)

  with $p_1, p_2, \ldots, p_m \in \mathcal{P}$
  unary-pred is a unary predicate over a primitive type of $\mathcal{PT}$
  pred is a m-ary predicate over $Pt_1 \times \ldots \times Pt_m$ *with* $Pt_i \in \mathcal{PT}$

The constraint have now been defined statically and its semantics should be clear. Subsequently, the operational behavior of them must be defined.

Chapter 5

Constraint Processing

This chapter describes the operational behavior of the constraints in the typology. In the first part the term *constraint processing* is described and some definitions are given revealing the benefits of the structured hierarchy for relaxation and propagation. Subsequently, the algorithm for processing is defined as a specialization procedure in the knowledge base and described for the various types of constraint. Vice versa relaxation as a generalization pro-

in the knowledge base is investigated in the next section. Only local mechanisms are provided here and the global aspects are discussed in section 7.3. Finally, some issues standing out the proposed methodology are summarized.

5.1 Concepts

The term *constraint processing* refers to three different operations:

1. **Constraint propagation** denotes the modification of constraints themselves. Originally, this term was introduced by Freuder [Fre78] to designate a technique of making constraints explicit already implicitly defined in the network by its definition. In our context, constraint propagation denotes a method of adding constraints, which were not in the network before, while the computation of a solution proceeds.

more direct restrictions between the variables of this constraint. On the other hand, the solutions of the constraint may be enhanced and thus allow more assignments for the affected variables. However, the set of additional values must be included in the initial domain of the variables. Both approaches are interchangeable in case of finite domains. Adding values to variables may enhance the set of possible solutions for the intensionally defined constraint and vice versa, increasing the set of possible solutions for the constraint, e.g. adding more solution tuples, allows more assignments for the variables as well, i.e. increases its current domain. [Fre89] discusses this point of view.

The viewpoint of *relaxation* is only focused on local states. Other authors use this term for referring to the CSP as a whole which can be solved with fewer backtracking steps than the original one, cf [Fre78].

Subsequently, though the objective for solving the CSP is also to find the "best" solution, which is in any case a global property, the algorithms only cope with local relationships due to the following reasons.

1. The approach for coupling structured hierarchies and constraints is basically incorporated for achieving local consistency in a weak sense: All known constraints at a time should be used as best as it can to eliminate assignments for variables which cannot contribute to any solution at all. Therefore, only the local *value propagation* process must be defined for that purpose. The final Backtracking also activates additional algorithms as *constraint propagation* but still only for deriving locally consistent states.

2. All processes for relaxation, though having a global point of view, require that the reasons for inconsistencies must be found locally. This information is then used to compute a globally consistent state, e.g. by retracting initial value restrictions. Thus, a local point of view is the basis for all relaxation methods.

3. Methods for relaxation are mostly domain or task dependent. They exploit additional information, like weighting the constraints [DL85] or defining a metric over the solution space [Fre89]. Consequently, the methods will be deferred until the task and the domain are discussed in more detail (cf chapter 6 and 7).

In order to explain the constraint processing over the structured hierarchy various terms must be defined. They are the fundamentals for revealing how the hierarchy is being used for propagation and relaxation.

Definition 5.1 *A* **gcs** *(greatest common subtype) $v \in \mathcal{ST}$ of two structured types $s, t \in \mathcal{ST}$ is defined as 'v isa s' and 'v isa t' and there is no common subtype $v' \in \mathcal{ST}$ of s and t with $v \neq v'$ such that 'v isa v''.*

Definition 5.2 *A class $v \in \mathcal{ST}$ is a* **rst** *(real supertype) of a type s if 's isa v' and there is a type $t \in \mathcal{ST}$ such that 't isa v' but not 's isa t' and there is an individual $ind \in \mathcal{IND}$ with 'ind isa v' but not 'ind isa s'.*

Informally spoken, the greatest common subtype defines the most general type in the set of structured types which is a subtype of both input types. The real supertype ensures that additional individuals are allowed afterwards by the last condition of definition 5.2. Although the condition '*ind* isa *v*' but not '*ind* isa *s*' requires that there must be a subtype with the

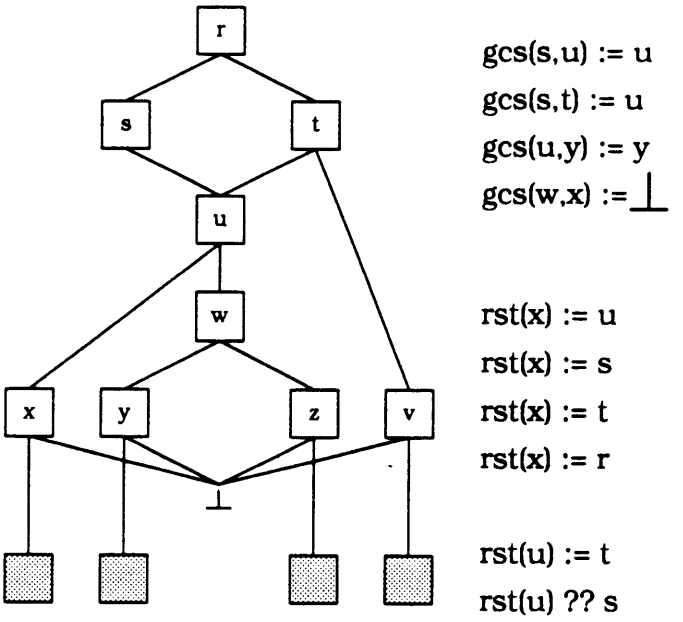


Figure 5.1: Greatest common subtype (gcs) and real supertype (rst)

properties of t—thus, the condition for t could be omitted–*rst* was defined this way in order to reveal how the algorithm for determining the required type will work: Test on the level of types and then on the level of individuals.

Figure 5.1 shows an example. $rst(x)$ yields all supertypes of x because each has subtypes *cl* for which it does not hold that '*cl* isa x'. If the last part of definition 5.2 would not be added, $rst(t)$ would yield r although this type does not provide additional individuals with respect to t. $gcs(w, x)$ results in $\perp$ because there is no common subclass of both classes. Obviously, the the *gcs* is uniquely determined whereas the *rst* of a class is not. Therefore, the algorithm for computing it is non–deterministic. Similar definitions apply for the greatest common subtype and the real supertype in $\mathcal{PT}$.

The computation of the *rst* is very exhaustive: First, search upwards for a superclass and second, searching downwards for new subclasses. In section 5.3 a heuristic is described which significantly prunes this search space.

5.2 Propagation

Value propagation is viewed here as a specialization procedure with respect that a more special subclass allows fewer assignments—the individuals—for a variable of the CSP than a more general class; hence, less search must be performed for finding a solution.

5.2.1 Specialization in the Knowledge Base

Algorithm 5.2 shows the computation of the most specialized structured class of two classes given as inputs. Obviously, the algorithm computes exactly the inheritance hierarchy as defined over the knowledge base. Consequently, if the two input types are structured classes in the knowledge base the result $r := \mathrm{u}\begin{bmatrix} p_1 & u_1 \\ \vdots & \vdots \\ p_n & u_n \end{bmatrix}$ of the algorithm and a structured class $\Lambda \ni r' :=$

algorithm *specialize(s,t)*
$u \leftarrow gcs(ST_s, ST_t)$
if $u = \perp$
  then return $\perp$
  else $\forall p_i$ $(1 \leq i \leq n)$
    $u_i \leftarrow gcs(p_i(s), p_i(t))$

$$\textbf{return } u \begin{bmatrix} p_1 & u_1 \\ \vdots & \vdots \\ p_n & u_n \end{bmatrix}$$

end **specialize**

Figure 5.2: Specialization of two structured types

$u' \begin{bmatrix} p_1 & u'_1 \\ \vdots & \vdots \\ p_n & u'_n \end{bmatrix}$ are related with each other such that $u = u'$ and r IS–A r', based on the fact that it is allowed to define a structured class in the knowledge base by overwriting the inherited values of its superclasses if this value is consistent with the ordering on the primitive types, i.e. a primitive type may be overwritten by one of its subtypes or by the constants of this type.

As a result, the algorithm does not need to compute the values of the parameters explicitly because they can be derived from the knowledge base if the two input arguments s and t are in the knowledge base as well. Hence, the loop in algorithm 5.2 can be omitted and only the $gcs(ST_s, ST_t)$ needs to be computed. This condition restricts the time parameter

constraints are evaluated. They must not be evaluated if the constraint does not result in a predefined object of Λ, e.g. in the example of figure 3.1 the class `Color-Monitor` is specialized with the current class of the variable *monitor* if the unary parameter constraint *type*(`color`) is activated but the constraint *size*(`19inch`) would not be evaluated that way because there is no corresponding class in Λ. If the parameter constraint does not result in an existing subtype the evaluation of the constraint is deferred and eventually applied to all remaining individuals. Instead, this constraint is only used as a predicate to check whether it can be fulfilled by the current type of the parameter later on, e.g. it checks whether the constant `19inch` is a constant of the current primitive type of the parameter. If the parameter does not fulfill this condition there is no individual possible which may provide the value `19inch`.

It is intuitively clear how this algorithm is used for class propagation. If an unary class restriction for the variable is applied, the current class of the variable and the requested new class are given as arguments to *specialize* and the result is attached as a new structured class to

algorithm *complete-specialization(s,t,Λ)*
if $s \in \Lambda$ *and* $t \in \Lambda$
  then return *structured class of* $gcs(ST_s, ST_t) \in \Lambda$
  else return *specialize(s,t)*
end **complete-specialization**

Figure 5.3: Specialization of two structured types with respect to a knowledge base

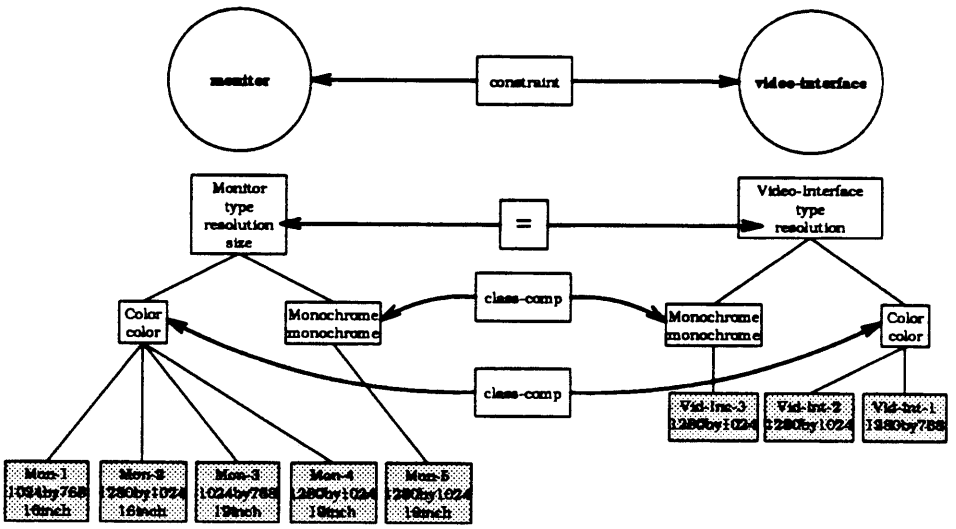


Figure 5.4: Example for value propagation

The knowledge base Λ is omitted in the following if it follows from the context. Also the algorithm will be named *specialize* unless otherwise mentioned

5.2.2 Value Propagation Methods

The value propagation process for the various types of constraints in the typology and the process as a whole can be declared together with the definition of the greatest common subtype and the specialization algorithm. The example in figure 5.4 is copied from figure 4.4.

Unary Constraints

Unary constraints restrict the values for one variable and have no direct effect for others. They may be added to the constraint network in two different ways:

1. They serve as an initial value restriction for some variables and thus, may represent additional requirements defining the CSP in more detail.
2. Unary constraints may be imposed by other constraints; in this sense being a result of constraint propagation.

The value propagation process for the three kinds of unary constraints is defined as follows:

- **class–rest**$(x: St)$

If the current class of variable x is St' then the new class of the variable will be *specialize*(St, St'). In case that $\perp$ is derived a relaxation must take place.

class–rest(*monitor*: `Color-Monitor`) would result in an assignment of `Color-Monitor` to the variable *monitor* with the initial class `Monitor`.

- **not–class–rest**(x: St)

  If the constraint is violated—the current type of the variable is St or one of its subclasses–a situation occurs where relaxation is required, cf section 5.3. Otherwise, no action need to be performed and the constraint is just attached to the variable to make sure that St can never be assigned to it. Consequently, a relaxation might still happen later on.

  not–class–rest(*monitor*: `Monochrome-Monitor`) would just be added to the initially assigned variable *monitor*. Later on the class `Monochrome-Monitor` or any of its subclasses would not be used to find an assignment for this variable.

- **param–rest**(**unary–pred**, $p(x)$)

  The current class of the variable may be St. If a class St' in the knowledge base exists which is defined as $St \wedge$ unary–pred($p(St)$) then this constraint is transformed into the equivalent constraint **class–rest**(x: St').

  If there is no such class, the parameter of the current class St is checked whether the constraint **param–rest**(**unary–pred**, $p(St)$) is satisfied. Again, if it is violated a relaxation must take place. There might also be a check introduced testing whether the constraint can be satisfied at all. This procedure is used by exploiting the fact that the hierarchy of the structured types is well–defined.

  If **param–rest**(=*color*, `monitor-type`(*monitor*)) is activated it would be immediately transformed into the equivalent constraint **class–rest**(*monitor* : `Color-Monitor`). In case that there is no such class, e.g. for class–rest(=19inch, monitor-size(*monitor*)) the constraint is attached to the variable and additionally checked whether the current value of the parameter is still capable of satisfying the constraint later on. Here, the test results in true because '19inch isa `Monitor-Size`'; thus, it may be provided by some individuals of the current class.

N–ary constraints

The constraints between classes are only defined as binary constraints. There is no loss of generality because n–ary CSPs over finite domains can always be transformed into binary CSPs with equal solutions. The classes in the hierarchy always represent a finite set of values. However, the constraints between the parameters may be defined as n–ary constraints because their domains may be infinite in some case, e.g. arithmetic constraints.

The value propagation for the n–ary constraints of the typology takes place by constraint propagation: New, unary constraints are added to the network as soon as one of the constraints between classes is activated.

- **class–comp**(x_1: St_1, x_2: St_2)

  If the current value of x_1 is St_1 then the current value of x_2 must be specialized with St_2 which is achieved by introducing a class restriction constraint for x_2, i.d. **class–rest**(x_2: St_2). The same mechanism happens vice versa.

class–comp(*monitor*: `Color-Monitor`, *video-interface*: `Color-Video-Interface`) works that way. As soon as a `Color-Monitor` is chosen a `Color-Video-Interface` is enforced as well.

- **class–incomp**($x_1 : St_1, x_2 : St_2$)

  If the current value of x_1 is St_1 the value of x_2 must not be St_2. Consequently, the constraint **not–class–rest**($x_2 : St_2$) is attached to x_2. Again, the same procedure works vice versa as well.

  class–incomp(*monitor*: `Monochrome-Monitor`, *video-interface*: `Color-Video-Interface`) may be added to the above example, though not providing any additional information for the constraint solutions. Now all individuals of `Color-Video-Interface` would be excluded as possible assignments for *video-interface* as soon as `Monochrome-Monitor` is chosen and vice versa.

- **param–comp**(**pred**, $p_1(x_1), p_2(x_2), \ldots, p_n(x_n)$)

  The evaluation of these constraints highly depends on the attached procedure for the predicate 'pred' which is defined for each primitive type individually. Therefore no general statement can be made about the constraint processing. An example should reveal some of the principles.

  The constraint **param–comp**(=, `resolution`(*monitor*), `resolution`(*video-interface*)) in the above example states that the parameter `resolution` must have equal values in both variables. However, the constraint has an additional functionality by exploiting the primitive types. Each time the parameter gets a more specific type or constant this value must be forced for the corresponding parameter in the other variable as well. For example, as soon as an individual of `Video-Interface` is chosen, therewith the value `val` for `resolution` is fixed, a constraint of the form **param–rest**(=**val**, **resolution**(*monitor*)) is attached to *monitor*.

  Other methods can be defined for each predicate differently. Consequently, it is even possible to introduce full arithmetic in order to allow also infinite domains.

Summarized it may be said that well–defined structure of the hierarchy allows a more constructive and functional approach to propagation whereas the conventional method would work destructively and predicatively on the same CSP.

5.3 Relaxation

In general, relaxation is required if the CSP is overconstraint and so no solution is possible satisfying the current set of constraints with tuples from the currently assigned values for the variables and may be regarded as being the contrary to the propagation process: A more general class must be selected in order to find individuals or structured classes which are consistent with the current constraints on that variable. As discussed in section 2.3.1 an overconstraint CSP may already be detected while computing a locally consistent solution. However, the search for a global solution via backtracking may still require a relaxation on account of the same situations:

- The propagation process computes $\perp$ as the greatest common subclass of two classes for a variable. In that case, either of the two classes must be retracted as assignment for the variable. This means on the level of the individuals that the intersection of the extension of both classes is empty.

- A constraint of the type **not–class–rest**($x : St$) is violated. Consequently, a superclass of the current type must be found which does not violate the constraint any more.
- Due to parameter restrictions no possible assignment of individuals to a variable is possible. In that case, either the constraint for the parameter must be weakened or other assignments of individuals must be found which do not violate this constraint any more. An assignment of more individuals requires to find a suitable superclass of the current class which provides these individuals as its extension.

Summarized it may be said that all situations require that a real supertype of the current type of the variable must be computed in order to overcome the inconsistent state.

In the above cases the naive relaxation method by retracting the constraint which locally causes the inconsistency [HGV+88] should not be taken into account as the sole possibility. However, this constraint must be compared with the other relaxation methods (retracting initial constraints, adding new individuals, etc.) in order to perform the "weakest" changes.

5.3.1 Generalization in the Knowledge Base

As mentioned above, the computation of the real supertype of a variable is very exhaustive. Therefore, a heuristic is introduced which neglects the last part of the condition in definition 5.2 "(*ind* $\in \mathcal{I}$ $\mathcal{ND}$ with '*ind* isa v' but not '*ind* isa s')" is omitted and only the types which have been previously assigned to the variable are taken into account. This is also semantically founded: There are only corresponding constraints in the network which enforced the assignment of one of the classes. Therefore, classes which occur in the hierarchy but have not been explicitly assigned to the variable before also do not have a corresponding constraint.

By consecutively applying *specialize* to the initial type s_0 of a variable the current type of a variable s_n is determined:

$$s_n \leftarrow specialize(s_{n-1}, specialize(s_{n-2}, \ldots, specialize(s_1, s_0)\ldots))$$

In the following, the set *structured types chain* is a representation of the temporal refinement of a variable's class.

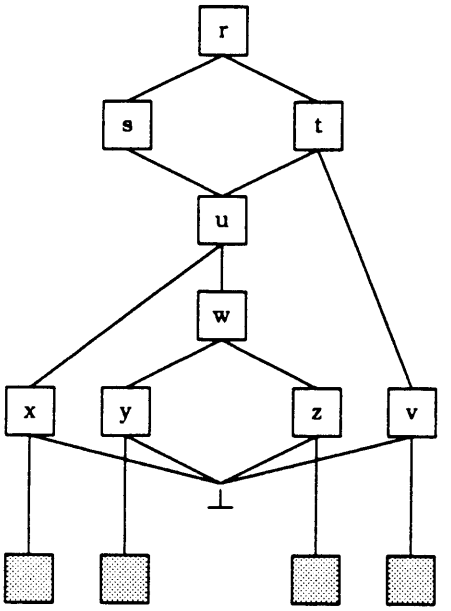
Definition 5.3 *A* **stc** *(structured type chain) of a variable x with initial type s_0 and current type s_n with the above defined relation between s_0 and s_n is the set*

$$\{s_n, s_{n-1}, \ldots, s_1, s_0\} \cup \{t \mid t \leftarrow specialize(s_i, s_{i-k}),\ 1 \leq i \leq n,\ 1 \leq k \leq i\}$$

The second set in the definition describes the structured types which have not been directly applied to the variable but have been computed with the *specialize* algorithm based on the lattice–like structure of the hierarchy. The example in figure 5.5 reveals the idea. The algorithm for computing a suitable class for relaxation uses the *stc* of the variable and works as in figure 5.6. The algorithm is called with two arguments. The *structured type chain* of the variable and a condition which must be an unary predicate applied to each class in *stc*. The predicate must be determined by the procedure which detects the local reasons for the inconsistency; thus, it may be a **not–class** constraint or a parameter constraint in case that a specific parameter restriction cannot be satisfied.

5.3.2 Selection of Candidates for Relaxation

The algorithm *ComputeCandidates* returns two sets as values. S_{fail} includes all classes which cannot be attached to the variable because they violate the condition *cond*. All these classes



$z \leftarrow specialize(z, specialize(t, specialize(s, r))$
$u \leftarrow specialize(s, t)$
because $gcs(s, t) := u$

$\Longrightarrow stc := \{z, u, s, t, r\}$

Figure 5.5: Example for the structured type chain (stc)

ComputeCandidates(*cond*, *stc*)
$S_{okay} := \emptyset$; $S_{fail} := \emptyset$
for $i := 1$ **until** n **do**
  $s := s_i \in stc$
  if *cond*(s) **then** $S_{okay} := s \cup S_{okay}$
    else $S_{fail} := s \cup S_{fail}$
$stc := S_{okay}$
return S_{okay} **and** S_{fail}
end **ComputeCandidates***

Figure 5.6: Computing of candidates for relaxation

must be deleted from *stc* and consequently, all constraints which forced these classes to be assigned to the variable must be deleted as well.

The second returned value describes the set from which the new current class of the variable must be chosen. Unfortunately, the ordering relation 'isa' on the structured types is only partial and therefore a lower bound of this set does not exist; hence, this class is non–deterministical. Obviously, for all classes which are in S_{okay} but cannot be compared with each other, a class in the hierarchy exists which is the greatest common subclass of both and therefore must have been previously computed; now being either in S_{okay} or S_{fail}. If it is in S_{fail} one of the uncomparable classes must be retracted as well. If it is in S_{okay} then it should be chosen as a new class for the variable because it is more special then the formers.

The set of classes which ultimately violate the constraints is now determined and the decision must be made whether to retract the classes in S_{fail} from the variable and the respective constraints or to deactivate the most recent constraint which was applied to the variable the time the inconsistency occurred.

As an example assume that a variable x with initial type r in figure 5.5 has the $stc := \{r, s, t, u, w\}$ and a relaxation condition which states **not–class–rest**($x : u$). *ComputeCandidates* yields $s_{fail} := \{w, u\}$ and $s_{okay} := \{r, s, t\}$ from which a new type must be chosen. It turns out that s and t are incomparable with each other and their greatest common subtype

is in s_{fail}. Consequently, either s or t must be moved from s_{okay} to s_{fail} as well and the other one is taken as a candidate for the new class of x. Assume this is t. Finally, all constraints which previously forced the assignment of the types in $\{s, w, u\}$ must be compared with the constraint forcing the relaxation. A measurement must be provided which now performs the "weakest" relaxation for the CSP, i.e. retracting either of the both sets of constraints.

So far the method only provides information based on a measurement for local decisions. However, the propagation process spreads itself out over the whole network. As a result, the reasons for an inconsistent state may be found in value restrictions for other variables. Unfortunately, the knowledge to provide a relaxation procedure from a more general point of view is domain and task independent. The approach in HieraCon is discussed in section 7.3 where two more global mechanisms are introduced which support the local decision for relaxation.

5.4 Summary and Conclusions

In order to complete the discussion of constraint processing by exploiting structured hierarchies the method is compared with the *conventional constraint processing* on the individuals level. This term refers to local propagation and backtracking as discussed in chapter 2. Also some additional benefits of the proposed methodology are summarized.

- The propagation is *constructive*; hence, no superfluous assignments are made for a variable which have to be retracted later on in any case.
- The constraints are partially *functionally* defined and propagated such that the processing methods becomes directed though the constraint can still be defined bidirectionally.
- The *maintenance* is simple because new individuals and classes may just be added to their "right" place in the structured hierarchy. Because the constraints are defined on a semantic level, e.g. the physical relationships between them are defined, the CSP still yields consistent solutions with respect to the new objects.
- Because a constraint typology was defined the *operational behavior* required by additional constraint, e.g. new predicates over primitive types, is known. Their operations must correspond with the other constraint methods in the typology.
- The *efficiency* is improved because whole sets of individuals are retracted by one step. Unfortunately, this advantage is partially destroyed because the use of HieraCon as a knowledge representation system rather than a constraint system requires additional expense for data management.
- The *data management* for storing the actions performed during constraint processing is minimized by using operations on classes as a whole.
- Although *relaxation* is one of the hardest problems in constraint systems, the approach provides a simple mechanism for computing local proposals for relaxation.

The first part of this work is now finished. The rest of the documentation reveals how the proposed method for constraint definition and processing are applied to a real-world domain: the configuration of workstations.

Part II

HieraCon

Chapter 6

The Configuration Task

So far, a specific kind of constraint satisfaction problems has been defined, cf definition 4.1, including a constraint typology and an accurate domain structure. An obvious question is whether this model has any practical application in order to prove its feasibility. One possibility—already used in the examples—is the configuration task as defined in this chapter. In the first section, a model for the configuration task is presented. Some deeper insights for specific configuration problems are discussed in the following. Finally, the additional requirements for modeling configuration as a CSP are listed which must be added to the presented CSP model.

6.1 Model

Felix Frayman in [FM87] defines configuration as a generic problem solving activity in terms of its inputs and outputs:

> *Given a fixed pre-defined set of components, an architecture that defines certain ways of connecting these components, and requirements imposed by a user for specific cases, either select a set of components that satisfies all relevant requirements or detect inconsistencies in the requirements.*

Some basic issues are intuitively included in this statement and influences the design for a constraint-based configuration system:

- The configuration task selects a set of components from a given set. It does not design any new artifacts. Therefore, the domain from which a solution must be derived is finite. Although other domains which describe the required functionality of the solution may be stated by constraints over infinite domains, e.g. arithmetic equations. Vice versa, the domain must be complete enough to provide the required functionality in principle.
- The input for the configuration task consists of three different kinds of knowledge.
  1. the fixed pre–defined set of components
  2. a description of valid connections between the components
  3. a description of the user requirements

A crucial part of this knowledge can already be represented in the discussed CSP model. The structured hierarchy holds the components, the constraints describe the architecture, and the user requirements are included as initial constraints. An objective of the further discussion is to elaborate these issues in more detail and to enhance the constraint solver appropriately.

- The solution of the configuration task—a selection of components—corresponds to a solution tuple of the CSP. Consequently, all components with equal functionality, e.g. all monitors, are represented with a single variable which provides this functionality to the overall solution.
- The term *relevant* implies that the requirements impose an order on the solution space or, from the user's point of view, are a means to describe preferences on how the system has to configured. If various components fulfill a required functionality, e.g. several monitors are finally left over, these criteria must select amongst them. The preferences even define what an *optimal solution* is: one which fulfills the most important or even all requirements as best as possible.
- The term *inconsistency* was already used to describe the situation in which relaxation is required, i.e. the problem is overspecified and no solution can be found. Consequently, a configuration system should not only detect the inconsistencies but also provide alternatives how to get a solution under weaker conditions or, in the specific case here, user requirements.

Some of the above issues can already be modeled in the CSP presented so far. Others require additional functionality on the one hand for the expressiveness of the representation language and on the other hand for the constraint solver itself. In the rest of this chapter these additional requirements are discussed.

6.2 Insights

Subsequently, some additional insights for configuration problems are introduced in order to provide a more technical point of view. Following this discussion, some additional requirements for HieraCon are introduced. Basically, the point of view in [FM87], [MF89], and [MF90] is adopted.

Functional Architecture

Reviewing existing configuration systems reveals that a typical configuration is based on the fact that not all the combinatorical possible architectures are taken into account. Instead, only configurations with similar *functionality* are investigated, e.g. if one would like to configure a "von–Neumann architecture" the required kind of components and its tasks in the overall system are pre–determined.

In terms of constraint reasoning, the layout of the constraint network states the composition of the solution and therewith the kind of components contributing to it. However, to recognize the functionality is more difficult. It is basically coded in the domain where all components providing the same functionality for the solution are comprised as individuals under one class. This class is then the initial domain of the corresponding variable. Later on, a meta–reasoning procedure which dynamically changes the layout of the network copes with this problem in

case that the individuals cannot be easily integrated in a single class. For example, the functionality "displaying data" may be overtaken by a simple ASCII terminal as well as by a high resolution graphics monitor. For obvious reasons, it is sensible not to represent this technically fairly different kind of components with a single variable. Contrarily, a system should not simultaneously consists of an ASCII terminal and a high resolution graphics monitor.

Key Components per Function

For each function there is a so called *key component.* This component determines the structure of the subtask for configuring this function. The other components in the subproblem are uniquely determined if the properties of the key component are fixed. Over and above it, key components are typically configured in the sense that user requirements are imposed on them.

For example, the monitor is typically described for designing the "displaying data" function. A user very rarely cares about the technical data of support components, e.g. video interfaces, cables, etc. Consequently, the architecture must be defined in a way such that the technical properties of the components can be derived from the description of the key component. But, the relations must be also be defined vice versa because sometimes support components impose restrictions on the key components as well. For example, some video interfaces may be incompatible with some workstations, but there is no direct constraint between monitors and workstation. Thus, as soon as a workstation is selected the remaining set of monitors is also reduced indirectly via the coupling through video interfaces..

There are two consequences for the underlying constraint solver:

1. The problem solving process is functionally driven from the key components of the subconfiguration via its support components to the overall solution. Consequently, the constraint processing gets a direction or even a complete inference chain starting from the user requirements on the key components to the support components.

2. At some point it may be required to make assumptions on selecting specific individuals or classes in order to proceed towards a solution of the CSP. In this case, the assumptions are made for the variables of the key components, so the support components are direct follow-ups in the search process.

Once again, this point of view correlates with the strict separation of constraints—represented as undirected relations— and their underlying operations which may be activated in a directed manner.

Reusable Components

Components may be used in two different contexts. They may either be reusable for different functions—temporally or serially—they play in a valid configuration or they cannot be reused by other components. From a constraint modeling point of view this problem can be easily tackled. If a component is reusable the variable representing this component has constraints to all subproblems in which it overtakes a function. Otherwise for each function a new variable with the same domain is created and added to the network.

An example may be the reusability of main memory for different software but the disk space can only be used sequentially.

Multi–function components

A special kind of components may have different tasks in different functions, e.g. a tape drive may be used for backup purpose as well as for software installation. Again, this can be easily modeled on the level of the network layout but the most difficult problem here is to find a solution with a minimal number of involved components but still fulfilling all the functions.

Optimization

One of the major differences between a constraint satisfaction problem and a configuration task is that the configuration problem is really an optimization problem whereas the CSP—finding any vital solution—is underlying and only a part of the process (see above). In general, global optimization is too difficult to cope with, especially, if it is tackled with constraint systems which usually make local decisions while on the contrary the optimization criteria refer to the overall solution.

6.3 Modeling as CSP

Some of the above mentioned problems can only be modeled very indirectly as a CSP. Consequently, some additional functionality must be provided.

- The issues "multi–function components" and "reusable components" require *additional representation* features. A language for describing the layout of the network and the explicit description of constraints between variables must be introduced. Therefore, the same classes of the network may serve as initial domains for different variables.
- The constraint network may change its layout *dynamically.* Additional constructs stating the situation in which a variable changes its state, e.g. may be deactivated and thus does not contribute to the solution any more, must be introduced and managed by the problem solver as well.
- The *relaxation* facility plays an important role for detecting the inconsistencies and providing support for finding weaker restrictions which still are suitable solutions for the user. So far, only a weak local relaxation mechanism is provided and therefore this issue must be elaborated in more detail in order to gain a more global point for view.
- *Optimization* criteria at least should influence assumptions which have to be made in specific cases, i.e. deciding which class or individual to select, relaxing user requirements, etc.

Some of the requirements are exhaustively discussed in the following chapter whereas others are only shortly mentioned.

Chapter 7

HieraCon

This chapter describes the system HieraCon (Hierachies and Constraints). The chapter starts with a summary of the characteristics of the system. Some additional features which are derived from the discussion of the last section are intensively investigated and tried to be modeled in the framework of a CSP. Subsequently, the modularization and the overall problem solving procedure are explained. Finally, some remarks about the implementation aspects are made, although no source code is discussed. Instead the principle features of Common Lisp and CLOS used here are explained and the characteristics of the modules based on the functionality they provide is presented.

7.1 Functionality

So far, the model of the CSP and the requirements of the configuration task have been discussed. Based on these issues the required functionality and also the characteristics of HieraCon can now be presented.

- The *propagation* process on the constraint level was already exhaustively discussed in chapter 5.
- *Relaxation* is one of the most difficult problems in constraint reasoning but it is also the

- Based on some properties of the configuration task it cannot be modeled with a sole CSP. One of the enhancements require that, in addition to the constraint process, a kind of *meta-reasoning* over the layout of the network must be performed.

- One of the major concerns was to design a system which is *modularized* on several levels. On the one hand side, a clear separation between the knowledge base and the problem solver makes the system usable towards an expert system shell approach [1]. On the other hand, in order to fully exploit the paradigms of object-oriented programming[2] the point of view that constraint-based modeling is "relations + operations" has proved its power.

These issues are the major points of concern in the rest of this chapter describing the functionality and the architecture of HieraCon.

7.2 Dependency Network

In order to provide the discussed functionality of the system, especially explanation and relaxation, a data structure must be introduced which stores the inference steps and keeps track of the changes in the variables. Typically these systems are called Reason Maintenance System (RMS) and have the task to keep the record of how a conclusion is reached. [SS89b] discusses several requirements for such a system.

As the coupling of a RMS and a constraint system was not the major issue of this work, basically a pragmatic approach was taken for such a system. Therefore, it cannot be directly compared with Doyle's TMS [Doy79] or de Kleer's ATMS [dK86], although some of their general principles are adopted. The most important one is the strict separation between the problem solver and the RMS. The problem solver must be able to run without the RMS and still draw correct conclusions. The only affect is that it is possible to retract previously made decisions non-monotonically.

Four major issues must be clarified to describe the behavior of the RMS:

- Type of the dependency information recorded
- Inconsistency management
- Type of information provided by the RMS
- Relationship with problem solving mechanism

The first issue was already intimated in section 5.3.1 where the structured type chain of a variable was defined. This is already a kind of dependency information which locally stores the modification of the variables' type over time. Additionally, it was mentioned that the constraints which force these type changes must also be detected but no mechanism was presented. The rest of this section outlines this and the other issues of the reason maintenance system in HieraCon.

[1] However, there was no time left to proof this claim by investigating a different domain.
[2] CLOS was chosen as implementation language, cf section 7.7.

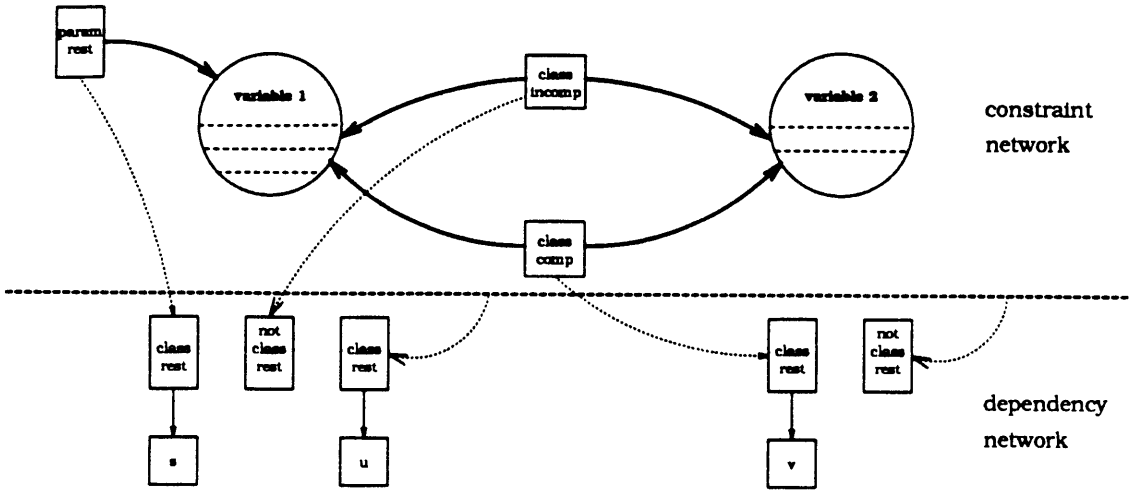


Figure 7.1: Recorded information for the variables

7.2.1 Recorded Information

Three major informations are stored for each variable.

1. structured type chain
2. the constraints activated on these variable, i.e. class restrictions, not–class restrictions and parameter restrictions
3. the correlation between them, i.e. for each type in stc the constraint which enforced it.

As these are only the local properties of a variable, i.e. the unary constraint activated for it, and in order to keep track of the constraint processing, the information which n–ary constraint imposed which unary constraints must also be stored in the RMS. Figure 7.1 should reveal the records. The constraint network is pictured in the upper part of the drawing, whereas the dependency network with its record is shown in the lower part. There are some links (dotted arrows) between the two parts which store the information from the constraint propagation process when new, unary constraints are attached to the variables. These constraints are assigned to the members of the stc ($\{s, u\}$ and $\{v\}$) which were derived as a result of applying them. The constraint in the upper left corner is assumed to be a user restriction and thus being added to the constraint network. But it is transformed into an equivalent class restriction constraint which is added to the dependency network as a processing step. The links from the dashed line to the unary constraints are assumed to be connected to other constraints in the network. The not–class restriction constraints does not have a corresponding class in the stc. Therefore, it is added to the variable without any further assigned information.

7.2.2 Inconsistency Management

The local inconsistency management was already described in section 5.3. With the data in the dependency network a more global technique is enabled. The concatenated inference steps become chains from the initial restrictions to the current type of the variable. Referring back to figure 7.1, some assumption are made. The constraints are defined as follows:
classes–incomp($variable_1 : t, variable_2 : v$)

class–comp($variable_1 : s, variable_2 : v$)
In addition, the initial parameter restriction forces class **s** to be assigned to $variable_1$. Consequently, there is a chain from the initial constraint to the unary constraint which forbids class **t** because the two defined constraints are consecutively triggered by classes **s** and **v**. This information can now be exploited as soon as the constraint **not–class–rest**($variable_1 : t$) is involved in any relaxation process and the information is provided that the constraint was eventually initialized by an initial parameter constraint. This issue is discussed in more detail in section 7.3.

7.2.3 Provided Information by the RMS

There are several kinds of information which can be provided by the RMS as the above picture already reveals.

- The newly created constraints are stored in combination with the variable they are applied to. In order to keep the system uniformly, this must be the constraints which are derived as a result of constraint processing as well as the assumptions (selection of classes of individuals) in form of constraints which are made while no more active constraints are to satisfy. This information is required to make sure that if a n–ary constraint is relaxed all consecutive unary constraints are retracted as well.
- The relationship between the types in the structured type chain and the constraints which enforced them is stored. The information is provided to the relaxation module in order to get the necessary information what constraints must be deactivated if a class is removed from the structured type chain.

Both information are only required if a relaxation takes place. Consequently, the paradigm that the inference engine is able to run without the RMS is fulfilled.

7.2.4 Relationship with Problem Solver

The problem solver—here: the constraint solver—and the dependency network are strictly separated from each other. The RMS gets the information about changes in the constraint

the inconsistent state—no more values are possible for a variable—of the CSP is detected at a specific location in the network, the actual cause typically is founded anywhere else. As the objective here is to model the configuration task (cf chapter 6) the objective is to reduce the reasons for the inconsistency to the initial restrictions for the possible values of the variables[3]—the user requirements. Consequently, there are non-local informations necessary and a global measurement should be provided to perform a relaxation. Additionally, this kind of knowledge provides rules of thumbs for the configuration task itself which can be represented in the knowledge base and used by the constraint relaxation algorithm incorporating these expertise. Two enhancements are presented:

1. The constraints are weighted according to their importance from the user's point of view as well as from the expert's viewpoint who defines the knowledge base.
2. A constraint on a higher level is provided which groups single constraints in the typology so far represented. If a relaxation procedure requires to relax such a constraint the constraint provides alternative solutions which are in some sense weaker than the original one.

These mechanisms are discussed in the following and the enhancements for the constraint solver is presented.

7.3.1 Weighted Constraints

According to [DL85] each constraint gets a value describing its relative importance for an overall solution. This value also corresponds to the relaxability of the constraint: Very important constraints are unrelaxable, important constraint are hard to relax, unimportant ones are easy to relax, etc. The mechanism is also a means for the user to express his preferences as mentioned in the model of configuration (cf chapter 6). For each initial restriction there is a corresponding weight describing how important it is that the constraint is satisfied by the final solution from the user's viewpoint. Consequently, the weighting function may provide an ordering over the solution space. [Sat90] makes a formal investigation of this point of view. In addition, there are some approaches to incorporate constraint hierarchies into CLP where the hierarchies are defined according to the importance of constraints [BMMW89].

In the following, the weights are taken as integers starting from 0 which denotes a very important or unrelaxable constraint. [DL85] introduces 10 different degrees for weighting the constraints. There are doubts that there is a cognitive adequacy for such a fine-grained measure. However, any kind of weighting function fits into the following scheme as long as it can be represented on the integer scale with 0 as lower bound. There is no need for an upper bound.

Constraint Representation

The syntax for the constraints is now enhanced in the following way. An additional argument is provided which represents the importance of the constraints. The same measurement is used for the initial restrictions, expressing the user's preferences as well as for the constraints in the knowledge base expressing the weight of constraints with respect to their technical importance. For example, the compatibility constraint between classes becomes now **class-comp**$(x_1 : St_1, x_2 : St_2, \textbf{weight})$ and similarly for all other types of constraints. In the following, this factor is called *relaxability* of a constraint.

[3] Provided that the knowledge base is consistent itself!

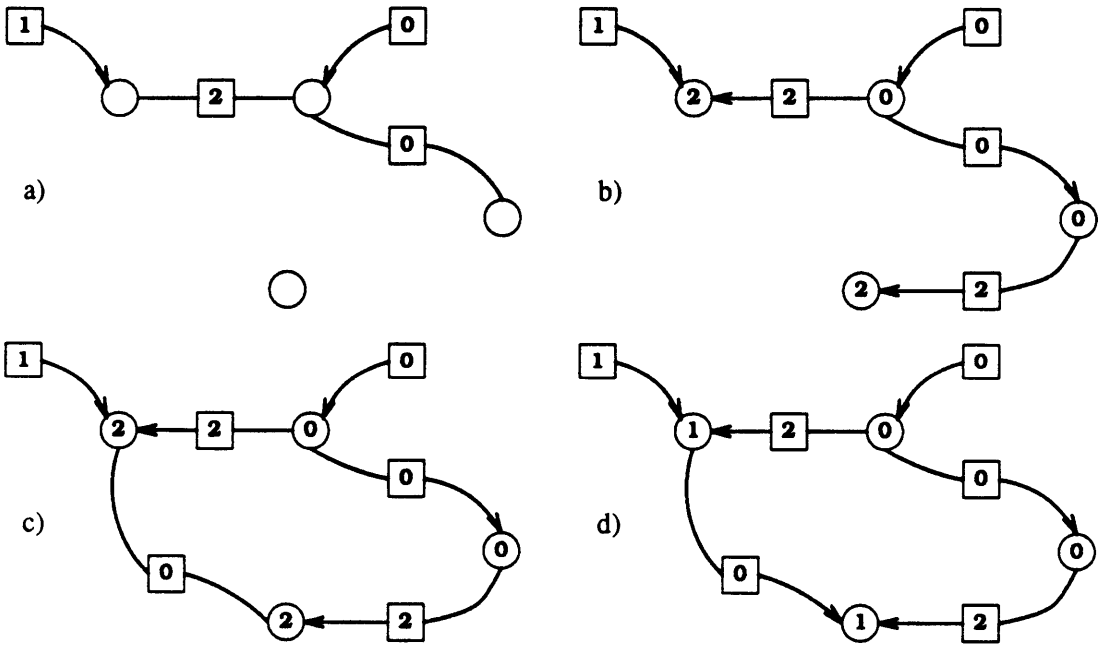


Figure 7.2: Propagating relaxability over the network

Constraint Processing

The constraint processing algorithm must also be modified in order to process the relaxability from the initial constraints to the constraints being dynamically added to the network. [FBMB90] discusses a similar approach.

For that purpose, the terminology of the relaxability of a variable is introduced. It expresses how easily the currently assigned class can be changed in order to find a more general one.

Definition 7.1 *The* **relaxability of a variable** x *(relvar(x)) is defined as* $\min_{weight}(cons(x))$ *where* $cons(x)$ *denotes the set of all currently active constraints of the variable.*

The initial relaxability of a variable is assumed to be ∞. Depending on the relaxability of the constraint the relaxability of a variable which is modified via that constraint is dynamically changed:

Definition 7.2 *Given are the variables* x *and* y *and a constraint between them with relaxability* r*. If this constraint affects the value of* x *while the current value of* y *activates it, the new relaxability of* x *is set to*

$$\min(relvar(x), relvar(y) + r).$$

Consequently, an assignment for a variable can never be relaxed once it has gotten the relaxability 0 because the function depreciates monotonically. Therefore if an inconsistency occurs on these variable no relaxation of the CSP is possible. The effect of propagating the relaxability from the initial constraints over the network is shown in figure 7.2. Variables are represented in circles whereas constraints are pictured in boxes. The numbers for the constraints denote the weight which was chosen for its definition. Numbers for variables denote their current relaxability, i.e. how easy the currently assigned class can be generalized by deactivating the constraint which enforced it. The picture shows the relaxability propagation in four steps.

a) The initial network consists of three variables, two initial constraints with relaxability 0 and 1, and two constraints in the network which are static in the sense that they are attached to the initial classes of the corresponding variables.

b) The unrelaxable initial constraint is activated and the corresponding relaxabilities for the variables are computed. As a side affect a new constraint becomes added to the network and propagates the relaxability to the previously unconstrained variable.

c) The second initial constraint is activated and as a consequence a new constraint is added to the network.

d) Two variables are affected and their relaxability must be computed according to definition 7.2. Therefore, its value is changed from 2 to 1 since the new restrictions are harder to relax and the minimum of both is taken.

With the above terminology it is now clarified how the locally computed sets of alternatives for relaxation is evaluated in order to find the weakest one. The example implies a heuristic which can be applied: The most important initial constraints should be processed first because they impose the least possibilities for relaxation or, in terms of constraint reasoning, constrain the CSP most. Unfortunately, as already seen in the example, the relaxability as a function of the constraints variable is not monotonically because constraints in the knowledge base can already be defined with an arbitrary relaxability. Therefore, the most recent constraint need not always to be also the easiest to relax.

As all constraints gets the additional relaxation factor, the ones in the dependency network in figure 7.1 are also enhanced that way. The set of all constraints which must be relaxed to resolve the inconsistent situation is explained in section 5.3.2. The computed set describes the classes which must be retracted as assignments for the variable if a relaxation takes place. The additional information how easy the corresponding constraints are to relax can now be used as a measurement for the weakest changes by comparing them with the constraints which actually caused the inconsistency. First, the definition of what a *weak relaxation* is provided.

Definition 7.3 *The* **relaxability of a set of constraints** C *(relaxa(C)) is defined as the minimum of the relexabilities of its constraints.*

Definition 7.4 *Given are two sets of constraints* C_1 *and* C_2. *A* **weak relaxation** *is a relaxation which deactivates all constraints in* C_i $(i = 1,2)$ *such that* $relaxa(C_i) < relaxa(C_j)$ $(i \neq j)$.

The decision of how to relax is based on this definition. The locally computed sets of constraints from the structured type chain and the assumed reasons for the local inconsistency can now be compared with each other on a global basis because the relaxability is propagated to each constraint dynamically. Figure 7.3 gives an example for the overall relaxation procedure. The additional information in the dependency network for the bounded

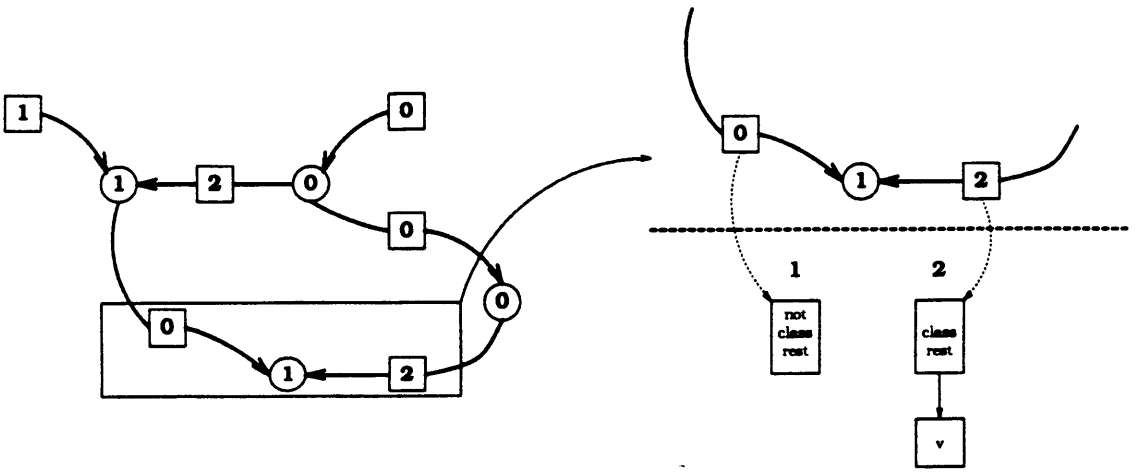


Figure 7.3: Example for the relaxation process

supposed to be the reason for the inconsistency shows that the weakest relaxation at that point is the deactivation of the constraint **class-rest**(v).

The changes have to be propagated through the whole network in order to keep the CSP consistent. Consequently, all constraints of which the deactivated one is the sole result are deactivated as well, but only as far as relaxable constraints are affected. In the above example only the deactivation of the immediate predecessor is performed because it has already been defined as relaxable in the knowledge base itself and there is no relaxation possible in the variable: its relaxability is 0.

7.3.2 Compound Constraints

In addition to the weight function which has revealed its power an additional means is presented which supports the internal relaxation of the knowledge. The idea is motivated by an example. The following knowledge must be expressed:

> *It is recommended to have LISP running with at least 16MB main memory; but 12 MB is the absolute minimum.*

This kind of knowledge is a typical rule of thumb used by the expert. From a constraint point of view it gives hints how the requirement of 16MB memory can be relaxed in case that it does conflict with more important preferences, e.g. price, etc. It can be regarded as an alternative constraint which can only be exclusively satisfied. For that purpose, *compound constraints*

- If the constraint is propagated try the "best" one first. In the above example, this is hard-coded in the relaxability of the constraint. Otherwise, this information can be detected from other user preferences, like price, performance, etc.

- If the relaxation procedure returns to such a constraint it deactivates the current one and tries the next one with lower relaxability. If there is none the complete constraint has to be relaxed according to the relaxability of the *XOR* if it is possible in the current context.

This would mean for the above example, that the first trial would be to add 16 MB to the configuration. If it is forced to be relaxed a new trial is made with 12 MB. If LISP was chosen with weight 0 than the selection is a minimal configuration and therefore unrelaxable. If LISP was selected as a relaxable preference then the complete constraint can be relaxed if 12MB is also impossible.

Two other compound constraints are added to the constraint typology as well:

- The compound *AND* constraint which states that as first trial all grouped constraints are activated simultaneously describing the best solution. If a relaxation of one of the constraints is required this constraints is removed and the other ones are not affected; the AND constraint is still satisfied.

- The compound *OR* constraint groups a set of constraints from which at least one must be satisfied. At first trial the one which is easiest to relax is added to the network and it is not a fault if other constraints are satisfied automatically by other propagation steps. If a relaxation takes place the single constraint is removed and it is checked whether another in the group is already satisfied. In that case no action is required. Otherwise the next constraint, measured on the base of its relaxability is taken and added to the network.

Obviously, the compound constraints can be intermixed with each other such that a complete boolean language is provided. Naturally, the unary *NOT* constraint would fit in this schema as well but is was already coded into the unary and binary constraints of the typology in section 4.3.

Although the grouped constraints need not to have any common semantics, i.e. they are constraints between the same objects, in general they have. In the example above they provide an alternative solution between the same classes. From a general point of view the compound constraints may not be regarded as complex constraints but as a means to describe the solution of a constraint by set composition, alternative sets and difference sets, cf [DP88]. Therefore, the relaxation process exploiting the semantics of compound constraints can be regarded as a procedure manipulating the solution sets of these constraints by various set operations.

7.3.3 Limitations

Although a couple of procedures are introduced to provide a whole variety of relaxation methods the mechanisms can only be regarded as heuristics in order to design an "intelligent" system. Therefore, the limitations of the methods have to be discussed.

The computation of the alternative sets for local relaxation in section 5.3.2 is based on two assumptions.

1. The most recent constraints applied to a variable the time the inconsistency occurred is a sensible candidate for relaxation. Especially, in case of parameter restrictions this may not be true because this constraint may have a lower relaxability than other parameter constraints before which excluded only a subset of the individuals compared to the set excluded by the most recent constraint. These other constraints are not taken into account by the algorithm because they need not necessarily enforce a class restriction. For example, a constraint with relaxability 1 excludes the remaining individuals $\{a, b, c\}$ and is taken into account as an alternative for relaxation, besides the constraints derived from the structured type chain. A previously activated constraint with relaxability 2 has excluded $\{a, b\}$ and is not taken into account, although, based on its higher relaxability it may even be a better candidate. To compute the optimal set of constraints to relax in this situation an exhaustive number of combinations[4] must be investigated at this point.

a complete logical description in the sense that the complementary set—intuitively described with the constraint—may not have a corresponding single class in the hierarchy[5]. The extension of this kind of constraints can only be decided on the individual level: It is

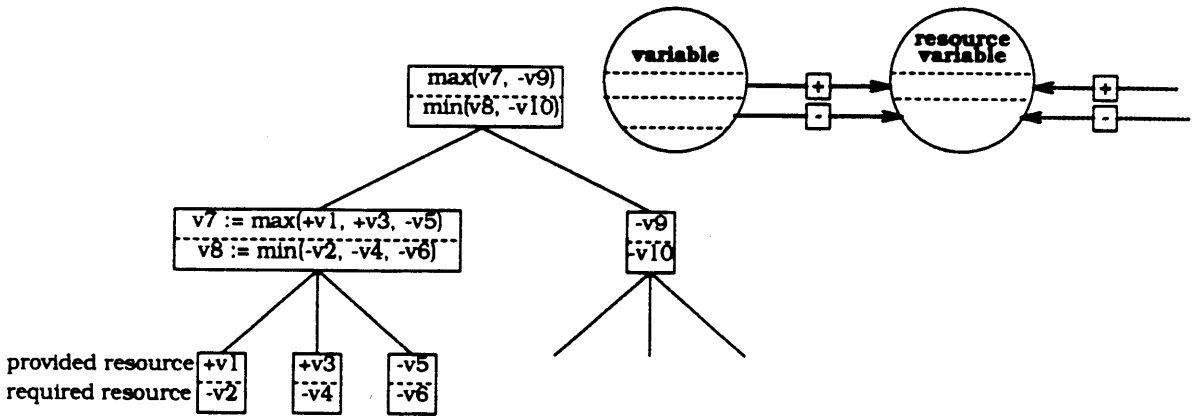


Figure 7.4: Exploiting resources for optimization purpose

task it is assumed that all optimization criteria are provided quantitatively; thus, there is no need to map from qualitative descriptions to quantitative sizes. These numbers are called **resources** in the following.

The individuals get a specific parameter which holds its resources. Also the classes get additional parameters with the same name and hold, in case of required resources, the minimum or, in case of provided resources, the maximum of the resources of all its direct subclasses or individuals. Figure 7.4 shows an example. The maximum/minimum of the resources of the individuals are attached to its direct superclasses which provide the information to its superclasses and so on. A special variable called the *resource variable* is added to the network. While the propagation process proceeds the variable get the values of all resources and sums them up. Consequently, in case of a required resource it holds the minimum of the resources which can be provided by all individuals still under consideration and vice versa in case of a provided resource it holds the maximum of these resources. The variable's update is performed dynamically at each time the value of a variable is refined. At that point the previous value is replaced by the current one. The value of the resource variable is monitored and as soon as a specific amount is exceeded or falls short (defined as a parameter restriction constraint on the resource variable) the remaining solution cannot fulfill the resource constraint.

Similar propositions as for the relationship between the locally consistent state and the global solution holds. If a state fulfills the conditions on the resource variable there may be no global solution at all for this restriction. For example, this methods sums up the price. As being a

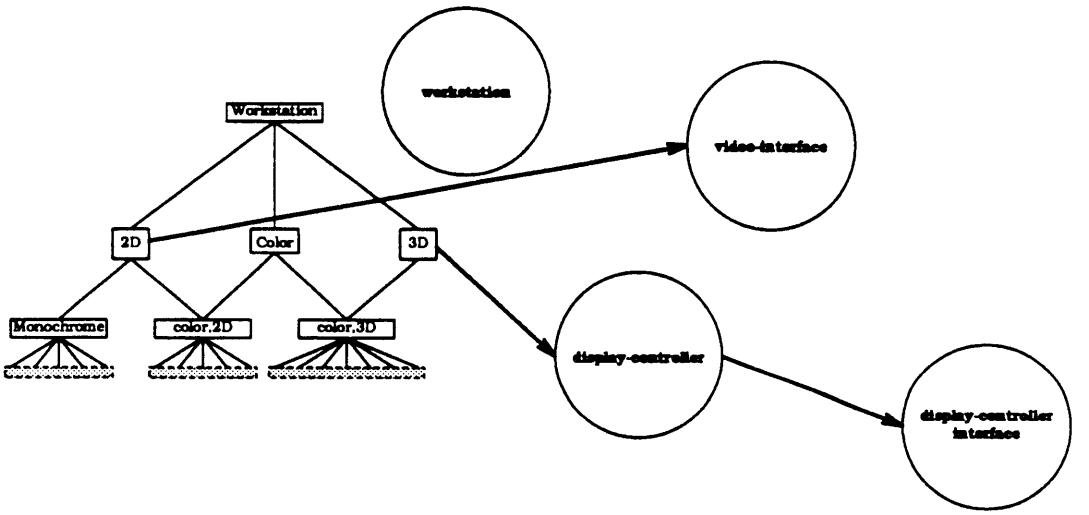


Figure 7.5: Example for a dynamically changing constraint network

performance" can be directly translated into a control information for the problem solver.

7.5 Network Layout

As mentioned in section 6.2 the configuration task requires that a more sophisticated language is provided to describe the layout of the network and to describe some dynamic changes performed while the propagation process proceeds. Variables must be connected in order to define constraints between them. In addition to this static description, variables must be activated and deactivated at run–time. An example is pictured in figure 7.5. It shows a network including four variables and the hierarchy for the class **workstation**. As it is seen the layout of the network changes due to the assignments for *workstation*. If the class **2D-workstation** or one of its subclasses is assigned a video–interface is required. Otherwise, if **3D-workstation** is a value of *workstation* a display–controller[6] is required instead. In addition, a component of class **display-controller** requires a display controller interface. Because **3D-workstation** and **2D-workstation** do not have a common subclass the two variables are exclusively activated. This kind of dynamic behavior must be introduced to the representation of the network layout. As a consequence, the solution of the CSP also varies dynamically. The size and composition of the solution refers to the variables finally being active.

From a more general point of view, this behavior can be interpreted as a simulation of a dynamic PART–OF relationship already mentioned in section 3.1.1 which describes the composition functions in the configuration task, cf section 6.2. In the example a fragment of the "displaying data" function is represented which consists of either a monitor (not introduced) and a video interface, or a monitor, a display controller and a display controller interface, exclusively. Obviously, this could have also been performed by summarizing the classes **display-controller** and **video-interface** to a common class but as they have different parameters because they have a physically different composition this approach would be very unadequate. The issue of reasoning over the layout of the constraint network is also discussed in [MF90].

The language for the layout of the network includes the following constructs:

[6] A display controller includes special hardware providing high graphical power, e.g. for running a 3D CAD system, whereas a video interface does not provide independent computational power.

- **def–var(x: `initial class`)** (define variable)

  A variable is defined and initialized with the most general class in the hierarchy it can adopt. Also it can be defined as being active from the very beginning, e.g. a workstation is required in any case.

- **group–vars($x_1, \ldots, x_n$)** (group variables)

  Variables are grouped together in order to specify between which variables the constraints are defined. This allows an addition of various variables with the same initial class, e.g. if a distributed system is configured a server workstation and the clients must be configured but both have as initial class **`workstation`**. Additionally, the problem of reusable components (cf section 6.2) can be represented by adding a variable to multiple groups.

- **def–key–var(x)** (define key variable)

  The key component property is modeled by labeling some variables as being key components. This information is exploited when an assumptions is made and was also discussed in section 6.2.

- **act–var–class(x, `class`)** (activate variable by class)

  A variable is activated as soon as a class is attached to another variable. The example above reveals the necessity of modeling this behavior.

- **act–var–var(x_1, x_2)** (activate variable by variable)

  The same situation on another variable in the network can happen because another variable is added.

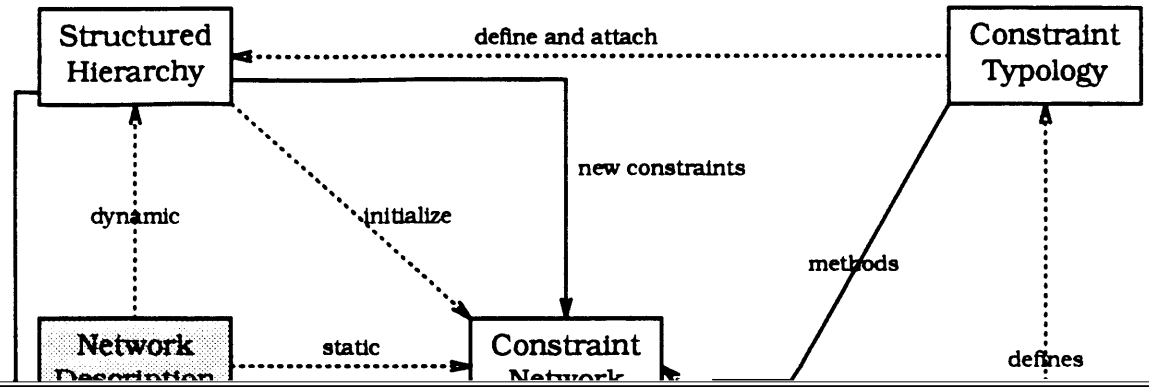
- **deact–var–class(x, `class`)** (deactivate variable by class)

- **deact–var–var(x_1, x_2)** (deactivate variable by variable)

With this notation the constraint network in figure 7.5 is modeled:

def–var(*workstation*: `workstation`)
def–var(*video–interface*: `video-interface`)
def–var(*display–controller*: `display-controller`)
def–var(*display–controller–int*: `display-controller-int`)
group–vars(*monitor*, *video-interface*)
group–vars(*workstation*, *video–controller*, *display–controller–int*)
act–var–class(*video–interface*, `2D`)
act–var–class(*display–controller*, `3D`)
act–var–var(*display–controller*, *display–controller–int*)

Unfortunately, this additional requirements enormously complicates the reasoning process. The RMS (section 7.2) must now also store information about the activation and deactivation of variables in order to undo such an action. It might happen that a class, which previously activated a variable, is later on retracted again. In that case the variable must be deactivated again as well. Another situation—a dynamically added variable cannot contribute to a solution—requires that a relaxation is performed for a class which previously activated this variable. All these information must be updated and kept consistently by the RMS in addition to its data management on the constraint level.


Structured Hierarchy
define and attach
Constraint Typology
new constraints
dynamic
initialize
methods
Network
static
Constraint
defines

the aid of the IS–A relation. In addition, the predicate constraints which are used as parameter restrictions must be defined appropriately and attached to its corresponding primitive types in $\mathcal{PT}$.

2. The next step is to define an architecture for connecting the components, describing a feasible solution by attaching constraints to classes in the knowledge base. This is performed by making instances of the constraints of the typology and of the predicative constraints of the primitive types.

3. With this knowledge a specific configuration task is defined by three different kinds of knowledge, pictured in shaded boxed.

   (a) The network layout is defined by the usage of the constructs in section 7.5. The static network description as well as its dynamic behavior, while variables are activated through classes, are described and added to the appropriate places. Herewith, the network is initialized with variables and its corresponding classes.

   (b) The user is now able to define additional restrictions on the constraint network representing his needs for the functionality of the system, e.g. what software should run on the system, what kind of monitor is preferred, as well as physical values, e.g. a workstation has been already selected and it should be enhanced to a complete system. These restrictions are added to the constraint network also in the form of constraints which are taken from the constraint typology as well. The quantitative restriction of resources are also in this category and they are added as constraints to the *resource variable*, cf section 7.4.

   (c) Additional requirements describing the optimization criteria from a qualitative point of view are also defined and provided to the constraint solver as additional information which can be used for making assumptions on the selection of components, e.g. the cheapest class/individual should always be taken, etc.

The configuration task is now defined and a solution can be computed.

7.6.2 Solving the Configuration Task

The main procedure of HieraCon is shown in figure 7.7. The set *Pending-Constraints* includes all constraints which have still to be evaluated. At the very beginning, these are all initial constraints of the network. At run–time the set is enhanced with all constraints dynamically being added by constraint processing. In case of a relaxation this set may also be altered because some constraints must not be evaluated any more because their activating values are retracted. The relaxation process is activated in two cases: Either an inconsistency is detected at a variable or the restrictions on the resource variable are violated. This loop is performed until all constraints have been evaluated and the current state is consistent such that is does not require a relaxation. It corresponds to a locally consistent state of the network (cf section 2.3.1) where no further local constraints can be applied and no inconsistency occurs. At his point assumptions must be made how to proceed. Any variable representing a key component is selected and further assignments are enforced by adding a corresponding constraint. This method roughly equals the backtracking algorithm (cf figure 2.1) which also assumes values for variables and checks its consistency in the current state with the difference that here the assumptions for attaching values to a variable are not based on a fixed sequence but on task/domain dependent knowledge. Summarized it may be said, that the two–stepped, conventional method for solving CSPs over finite domains (local consistency + backtracking)

```
Solve(Initial-Constraint-Network)
Pending-Constraints := active-constraints(Initial-Constraint-Network)
repeat select one constraint                                    ⟸1.
       if the constraint is not satisfied
         then satisfy it
       if inconsistent/non-optimal state
         then relax network                                     ⟸2.
       Pending-Constraints := active-constraints(Network)
       if Pending-Constraints = ∅
         then if solution found
           then return solution
           else if no solution possible
                  then explain no solution
                  else Pending-Constraints := assumptions       ⟸3.
       until Pending-Constraints = ∅
end *Solve*
```

Figure 7.7: Principle algorithm for solving the configuration task

is reflected in this algorithm as well. The major difference is that the procedure taken here is knowledge–driven and not purely data–driven.

The three crucial places where heuristics play an important role are marked in the algorithm with an '$\Longleftarrow$'.

1. The constraints with the lowest relaxability should be selected first because they impose the hardest relaxable restrictions on the network. One expectation is that therewith overconstraint problems can be detected early. Additionally, constraints over classes are preferred to those which are value restrictions on parameters because the whole processing is based on the benefits of these kind of constraints.

2. The relaxation is based on several heuristics which were exhaustively explained in section 7.3. In addition, user feed–back can be included here which is intimated in figure 7.6 by an arrow from the "user" to the relaxation module.

3. Additional assumptions made by the problem solver are based on two heuristics. First, the requirements define an order on how the alternatives are selected, e.g. "take always the cheapest component". This is only a locally optimal decision and obviously needs not necessarily lead to a globally optimal solution as well. However, together with the resource variable it overcomes its strict locality with the following observation. As a new assumption is made also new constraints can be derived from the knowledge base. Evaluating these constraints leads to further restrictions in other variables. Monitoring the changes on the resource variable gives a hint whether its value is refined towards the intended direction. For example, (cf figure 7.5) if the cheapest workstation is chosen and therewith only a video interface and not a display controller is possible, the performance for graphical operations will also decrease which can already lead to a retraction of the assumption at that point that the cheapest workstation should be taken although a great part of the search space, e.g. selecting a printer, disk cartridges, etc. has not even be touched. Second, the idea of key components restricts the number of variables which have to be taken into account for refining the values assuming that by attaching values to these variables the others are automatically determined by constraint processing. Also a user feed–back on how to proceed may be introduced here.

Again, the heuristics may serve as a support to propose continuations for the user at any point or they can be applied fully automatically. However, they need to be refined by investigating a real world application and also by taking into account the expert's approach to the configuration problem.

7.7 Implementation

The modules of HieraCon have been outlined on a structural and functional level. This section elucidates some of the implementation aspects. So far, a kernel of the described system has been implemented.

7.7.1 Language

Although Common Lisp with CLOS [Ste90] was chosen for some environmental restrictions it has proven its feasibility because this language includes several programming paradigms reflected by the component's functionality as shown in figure 7.6.

Structured Hierarchy

The structured hierarchy is an *inheritance hierarchy* and therefore implemented in CLOS. The objects in the structured hierarchy are defined by a CLOS "defclass" function automatically providing class and slot access functions.

The major functionality of the hierarchy for the problem solver is the computation of the greatest common subclass of two classes and the general IS–A relationship which is automatically provided by the Lisp "subtypep" function because CLOS introduces a new type to the hierarchy of Common Lisp types for each class. As there is no direct way to access to the properties of a class, e.g. default slot values, a prototypical instance of a class is assigned to a variable. In order to get the direct sub– and superclass of a class the non–standard CLOS functions "clos:class–direct–subclasses" and "clos:class–direct–superclasses" have to be used.

Also the constraints between the classes are prototypical instances of an object in the constraint typology. Another instance of the same class is then attached to the variables as soon as these variables are grouped with each other. Therefore, several variables with the same initial class can be defined.

Constraint Typology

The classes of the constraint typology which is also enhanced by the compound constraints

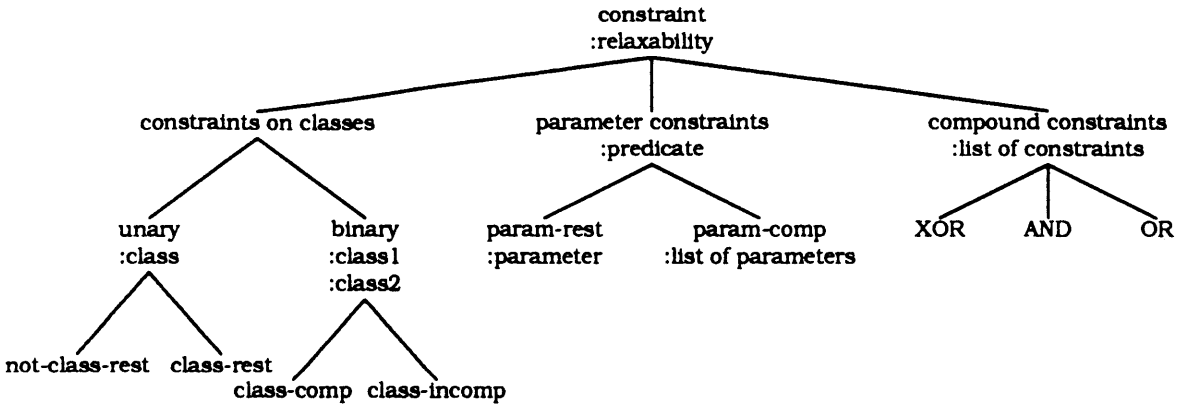


Figure 7.8: Constraint Typology

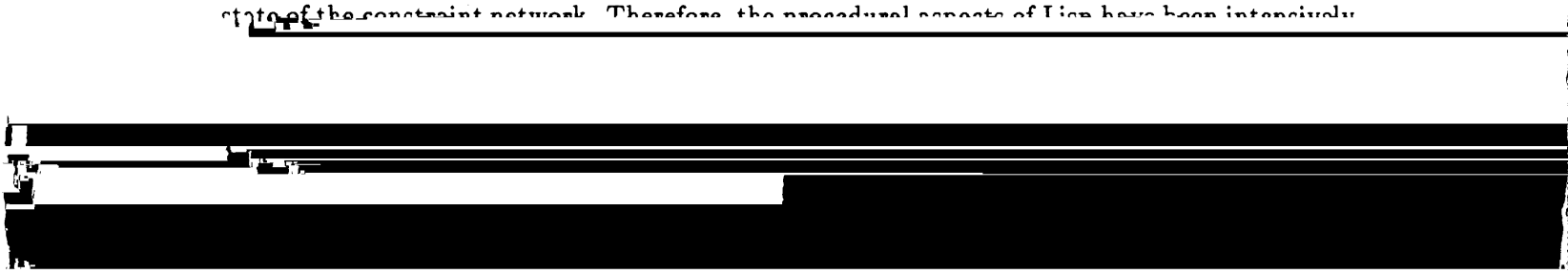
- a method for relaxing a constraint, i.d. handing its negation as the relaxation condition to the module
- a method which undoes a previously performed action on a variable

Also the ":predicate" slot of the class "parameter constraints" is filled with a predicate also defined as a method attached to the primitive types which are its arguments.

Consequently, the it object oriented CLOS–feature *generic functions* can be fully exploited for this module.

Problem Solver

The basic algorithm of the problem solver is shown in figure 7.7. It is obvious that is has mainly *procedural aspects.* The main loop, which is also the major control structure of the whole systems, is controlled by a set of constraints sorted by relaxability and their type as sort key. The algorithm activates the modules for constraint processing depending on the current state of the constraint network. Therefore, the procedural aspects of Lisp have been intensively



exploited for this module.

Its input are the structured hierarchy, the constraint network and the constraint typology. The output is a solution for the defined problem. Embedded are the constraint propagation and the constraint relaxation module requiring for information on the current state of the network.

- a predicate which checks the consistency of a variable's value
- a method which adds and deletes constraints

Dependency Network

The dependency network includes the RMS and is also *functionally* realized by providing several methods:

- add a new constraint to the network together with its attached class
- delete a constraint and undo modification it enforced
- a method which computes the candidates for relaxation from the structured type chain

Also, a *functional* point of view on this module seems to be appropriate using this Lisp paradigm.

7.7.2 Current State

To date some major aspects for proving the feasibility of the ideas has been implemented.

Chapter 8

Existing Systems

This chapter tries to put HieraCon into relation with five other already existing systems. One—ACK—is merely a programming tool but was implemented in CLOS as well and so gives some useful hints of how to cope with specific CLOS restrictions while implementing a constraint system. The others are also complete configuration systems—COSSACK—or shells—IDA, PLAKON, Platypus—with a functionality corresponding to the presented system. Vice versa, these systems also influenced the conceptual structure of HieraCon. Only the cited literature was taken into account; meanwhile, the systems may have been continuously developed further on. The lack of describing a feature was not taken to be a hint that the functionality is not provided. Additionally, only those features are intimated here which shows similarities to HieraCon.

8.1 ACK

ACK (a *C*onstraint *K*it) [LDKT88] provides a collection of classes, methods and functions allowing to define constraints between instances of CLOS [Ste90] classes. Therefore, it may be regarded as an enhancement of the object–oriented paradigm of CLOS in order to define also constraint–style relationships such that they are invertible in contrary to ordinary Lisp functions. It is intended as well to establish constraint primitives which can be inherited by other CLOS classes to provide a platform for building more complex constraint mechanisms on the basis of ACK. Its functionality is motivated by the design task rather than the configuration task. Therefore, the methods emphasize the point to propagate values actively from one variable to another.

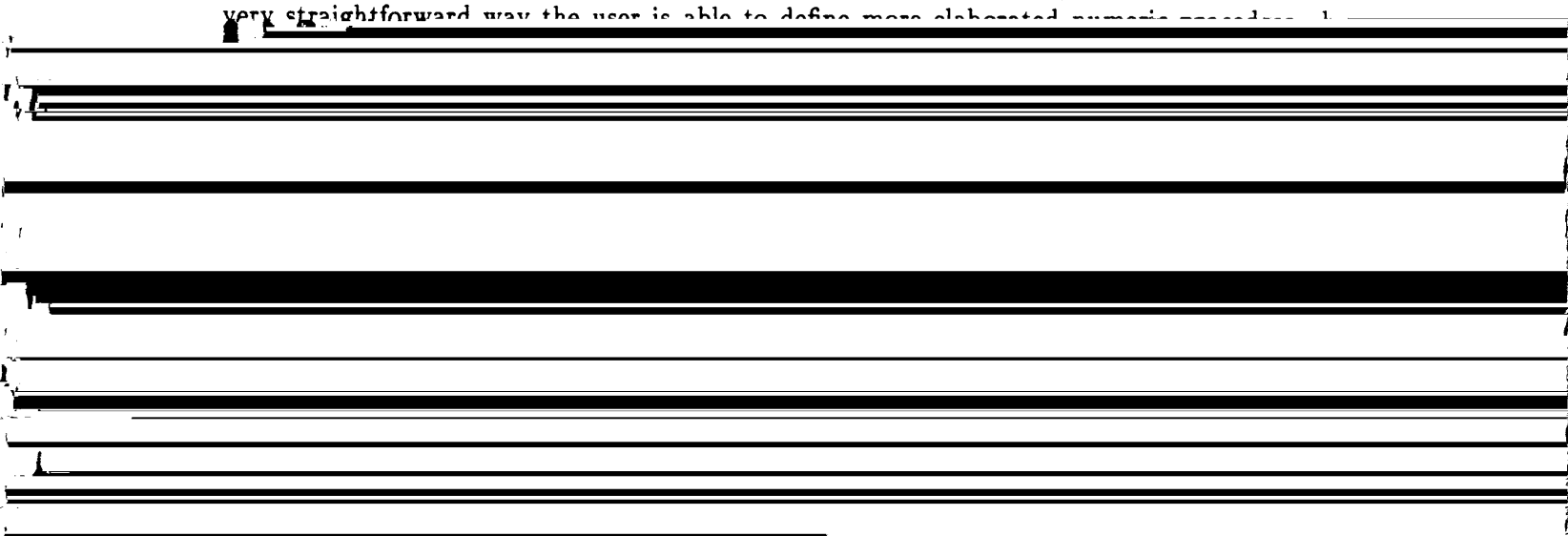
The classes *constrained-object* and *alg-constraint* are provided and must be defined as super-classes of the user–defined classes such that the constraint methods are inherited. In general, classes are additionally structured by having instances of other classes as values for their slots which also might have slots and so on.

```
(slot-value
 ...
 (slot-value
  (slot-value my-class 'var-1)
  'var-2)
 ...
 'var-n)
```

is an example for such a "nested" data structure. For the purpose of accessing a slot in a nested structure the notation of a path is introduced which references to a slot by denoting all the slot names of all included objects, e.g. (`:path <var-1> <var-2> ... <var-n>`). In HieraCon the objects are assumed to have a flat structure (primitive types as slot values).

Various functions are defined to manipulate the values of the slots, accessed by paths, which serve as variables. The value of slots can be actively determined by forcing all constraints referencing to that slot to become active. As each constraint is taken into account individually this procedure corresponds to computing the locally consistent solution for a variable.

The second class *alg-constraint* copes with algebraic constraints such that equations between terms over real numbers can be defined between classes. However, the constraints are based on Lisp functions and do not enable a handling of equations on a symbolic level. But in a very straightforward way the user is able to define more elaborated numeric procedures



exploiting pre–defined "hooks", ":before" and ":after" methods.

Methods for defining compound constraints and relaxability are already discussed in the document an can be easily introduced. Summarized it may be said, that ACK provides some features on the CLOS level which are also required in HieraCon; thus, it gives some hints of how to include constraints into a hierarchy of objects (CLOS classes).

8.2 COSSACK

COSSACK [FM87] is a constraint–based configuration system from which the model of configuration (cf chapter 6) was derived. Therefore, it sets value on the understanding of user needs and optimization criteria, too. Its application domain is the configuration of micro–computer systems. The major point of view in HieraCon that configuration modeled as a CSP includes constraint satisfaction and a meta–reasoning over the layout of the network can be found as well [FM87]:

> *A generalized constraint reasoning problem involves two interwoven search spaces: first—a space defining relevant variables and constraints identifying a standard constraint satisfaction problem, and second—a space of solutions to the defined CSP problem or finding a consistent assignment of values to the variables which in turn changes the first space.*

The second difference to a CSP is also addressed as *"the configuration problem is really an optimization problem"*

criteria are only included for making local decisions. The implementation is performed in a tool for building domain–independent, knowledge–based systems for design problems.

8.3 IDA

IDA [Pau90], an expert system shell for modeling the concept phase in construction problems, includes a constraint module for defining relations between plan variables. Its knowledge in the domain hierarchy (*static knowledge base*) is structured by two relations: the IS–A and an enhanced PART–OF relation such that solution alternatives can be represented statically and control the inference process.

The constraint system is closely coupled via the constraint variables with an ATMS which is only capable of storing directed dependencies whereas the constraint system represents undirected dependencies. It is also organized as a typology with various attached methods to each constraint class.

The solution of the CSP is derived by using the local propagation method (cf section 2.3.1). The values of variable assignments are chronologically stored with each variable, similarly to the structured type chain. However, the propagation procedure works destructively. Reasons for an inconsistency are locally derived on similar assumptions and, by exploiting the information stored in the the ATMS, the necessary backtracking steps are performed. Additionally, constraints being already implicitly defined between non–neighbored variables are made explicit and become added to the network such that later on the same inconsistency will not occur again. For controlling the relaxation process by retracting constraints, the constraints are weighted. The information is used to define a metric on the CSP. It locally decides the weakest relaxation by backpropagating constraints to their assumed reasons which are either assumptions in ATMS nodes or technical requirements defined in the static knowledge base.

8.4 PLAKON

PLAKON [CGS89] is an expert system shell for planning– and configuration tasks. An outstanding property is the feature of providing various *modes* for controlling the inference process (backtracking–free, intelligent backtracking, TMS supported mode, and an ATMS supported mode). A so–called *dynamic knowledge base* stores all dynamically created objects whereas the *static knowledge base* holds taxonomical knowledge, based on the IS–A, and a compositional hierarchy based on the PART–OF relationship. A classifier for automatically introducing new objects into the static knowledge base is provided to support knowledge acquisition and knowledge base maintenance. Here, the major difference to HIERACON is that the PART–OF relationship is modeled in the static knowledge base and not in the layout of the constraint network itself.

Constraints in PLAKON are used to represent and evaluate dependencies between objects. They may either constrain properties or make statements about the existence or non–existence of objects. Therefore, they roughly correspond to classes in the constraint typology of HIERACON with "constraints on classes" and parameter constraints. PLAKON also provides generic types

case that multiple variables with the same domains exist. The constraint definition is pattern based. A precondition is added to the constraint in form of a pattern which fixes the situation the constraint can be activated.

The problem of coping with resource constraints with an arbitrary number of input arguments, in HieraCon modeled with the resource variable, is here represented with constraints having a variable number of arguments, so-called "set–valued" constraints. The predicates over primitive types provide a more uniform approach to non–standard data structures, e.g. intervals, complex numbers, in constraints than the approach taken in PLAKON because here conceptually, there is no difference while defining or propagating such a constraint. PLAKON provides different algorithms for all of these constraints and does not assume a well-defined type hierarchy. Relaxation factors for weighting the constraints and the compound constraints (here: constraint bundles) are also introduced but they cannot cope with the relaxation of class or not–class dependencies.

8.5 Platypus

Platypus [HR89] is an expert system shell modeling synthesis and diagnosis problems as constraint–based recognition tasks. It basically allows an incremental refinement of the solution description where only the necessary incremental amount of work is done at each step.

Configuration in Platypus is based on the same model of configuration, cf chapter 6. Its major technologies involved are a model–based knowledge representation (schemas), a rule processor engine (horn logic rules driving the reasoning process), a constraint propagator engine, and a truth maintenance system (for exploring alternative search paths or network inconsistencies). The TMS supports backtracking for both components, the constraint propagator and the rule engine. Constraints are merely defined between parameters of the objects. The constraint propagation algorithm is derived from [MMH85] as discussed in section 2.3.2. There is no method introduced to define preferences over constraints. Instead, an ordering among the possible values of a parameter can be defined which controls the sequence they are selected for completing the solution.

Chapter 9

Conclusions

HieraCon so far discussed on a conceptual level need to be further developed and implemented. It should have become clear how the three major point of views mentioned in the introduction

Constraints = Relations + Operations
efficient solver for search problems
Constraints describe local relationships between entities.

have been exploited to design a constraint–based configuration system. Based on the intended functionality and characteristics of HieraCon (cf section 7.1) some final conclusions are drawn here.

- The *propagation* process on the constraint level was defined to exploit the structure of the domain intensively. Typically, classes represent as set of individuals with common properties. However, not all values for parameters are represented as an own class. Therefore, a mechanism may be provided which recompiles the hierarchy for solving subsequent configuration tasks by introducing new classes representing specific parameter restrictions which may have frequently occurred in previous sessions.
- The *relaxation* procedures are basically heuristics applying static domain dependent knowledge to the inference process. Additionally, the weights of constraints are manipulated. The question is whether they keep their declarative meaning with the computation rule in definition 7.2. This issue can only be answered through a pragmatic analysis of configuration session applying the proposed methods.
- A sophisticated *explanation* becomes extremely important through the inclusion of user–feed back for controlling the problem solving. Consequently, not only final solutions (non–existence of a solution) must be explained but also questions of the kind "How does the performance change if I select a 3D workstation?".
- Finding an optimal solution was reduced to fulfill number restrictions on resources. However, the term *optimization* is not even clear from a cognitive viewpoint as humans even cope with contradictory criteria such as low price and high performance and still find an optimal solution from their point of view. Consequently, a model should be developed to cope with inconsistent user preferences in an adequate way.
- One of the outstanding properties of HieraCon is that the configuration task although

framework for CSPs: dynamic CSPs over structured domains. This issue should be further on developed in order to incorporate other CSP techniques and algorithms as shortly mentioned.

The *meta-reasoning* feature over the layout of the network also shows that an enhanced methodology may incorporate features which are derived from the task specific requirements very naturally.

- The modularization of the whole system, especially the constraint typology makes it possible to introduce more sophisticated constraint types, e.g. constraints which can handle equations over real numbers symbolically, as long as they can provide the methods for the constraint processing.

Not mentioned features are additionally imaginable, like an ATMS based RMS for investigating several configuration simultaneously, operations research methods for providing sophisticated optimization procedures, etc. However, even the features presented so far are not proven to be feasible as long as they cannot be applied to a real-world knowledge base (which is the lack of many AI projects) provided by configuration experts.

Bibliography

[AKN86] H. Aït-Kaci and R. Nasr. LOGIN: a logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185–215, 1986.

[BBK+91] A. Bernardi, H. Boley, C. Klauck, P. Hanschke, K. Hinkelmann, R. Legleitner, O. Kühn, M. Meyer, M.M. Richter, F. Schmalhofer, G. Schmidt, and W. Sommer. ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge. In *Proc. 11th International Workshop on Expert Systems & and their Applications, Avignon, France*, volume 1, pages 133–145. EC2, 1991.

[BFL83] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. Krypton: A functional approach to knowledge representation. *IEEE Computer*, 16(10):67–73, 1983.

[BH91] F. Baader and P. Hanschke. A scheme for integrating conrete domains into concept languages. In *Proc. 12th International Joint Conference on Artificial Intelligence*, 1991. forthcoming.

[BLR89] R.J. Brachman, H.J. Levesque, and R. Reiter, editors. *Proc. 1st International Conference on Principles of Knowledge Representation and Reasoning.* Morgan Kaufmann Publishers, Inc., 1989.

[BMMW89] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint Hierarchies and Logic Programming. In *Proc. of ICLP 89*, pages 149–164, 1989.

[Bra83] Ronald J. Brachman. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16(10):30–36, 1983.

[BS85] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[CdPV89] Th. Christaller, F. di Primo, and A. Voß. *Die KI-Werkbank BABYLON.* Addison Wesley, 1989.

[CGS89] R. Cunis, A. Günter, and H. Strecker, editors. *Das PLAKON-Buch*, volume 266 of *Informatik Fachberichte.* Springer-Verlag, 1989.

[dK86] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[DL85] Y. Descotte and J.-C. Latombe. Making Compromises among Antagonist Constraints in a Planner. *Artifical Intelligence*, 27:183–217, 1985.

[Doy79] J. Doyle. A Truth Maintenance System. *Artificial Intelligence*, 12:231–272, 1979.

[DP88] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[FBMB90] B. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33(1):54–63, January 1990.

[FM87] F. Frayman and S. Mittal. COSSACK: A constraints–based expert system for configuration tasks. In D. Sriram and R.A. Adey, editors, *Knowledge Based Expert Systems in Engineering: Planning & Design*. Computational Mechanics Computation, 1987.

[Fox86] M. Fox. Observations on the Role of Constraints in Problem Solving. In *Proc. 6th Canadian Conference on Artificial Intelligence, Montreal, 1986*, pages 172–187. Presses de l'Université de Québec, May 1986.

[Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Fre82] E. C. Freuder. A sufficient condition for backtrack–free search. *Journal of the Association for Computing Machinery*, 29(1):24–32, 1982.

[Fre89] E.C. Freuder. Partial Constraint Satisfaction. In Sridharan [Sri89], pages 278–283.

[Güs89] H.-W. Güsgen. *CONSAT - A System for Constraint Satisfaction*. Pitman, London, 1989.

[HGV+88] J. Hertzberg, H.-W. Güsgen, A. Voß, M. Fidelak, and H. Voß. Relaxing Constraint Networks to Resolve Inconsistencies. In *Proc. of GWAI '88*, pages 61–65, 1988.

[HPC90] HP 9000 Workstations — Configuration Guide. Hewlett–Packard Company, Product Number 5954–8594, January 1990.

[HR89] W.S. Havens and P.S. Rehfuss. Platypus: a Constraint–Based Reasoning System. In Sridharan [Sri89], pages 48–53.

[JMSY90] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. 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.

[Kas89] S. Kasif. Parallel solutions to constraint satisfaction problems. In Brachman et al. [BLR89], pages 180–188.

[Kum90] V. Kumar. Algorithms for constraint satisfaction problems: a survey. Technical Report ACT–RA–041–90, Microelectronics and Computer Technology Corporation, Austin, TX, February 1990.

[KY89] N. Keng and D.Y.Y. Yun. A planning/scheduling methodolgy for the constrained resource problem. In Sridharan [Sri89], pages 998–1003.

[LDKT88] M. Lemon, J. Dailey, A. Kuchinsky, and I. Tou. ACK (A Constraint Kit). Technical Report STL–TM–88–06, Hewlett-Packard Laboratories, August 1988.

[Lel88] W. Leler. *Constraint Programming Languages - Their Specification and Generation*. Addision–Wesley Publishing Company, 1988.

[Mac77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mes89] P. Meseguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–17, 1989.

[MF87] S. Mittal and F. Frayman. Making partial choices in constraint reasoning problems. In *Proc. of the 7th National Conference on Artificial Intelligence, Seattle*, pages 631–636, 1987.

[MF89] S. Mittal and F. Frayman. Towards a generic model of configuration tasks. In Sridharan [Sri89], pages 1395–1401.

[MF90] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *Proc. of AAAI 90*, pages 25–32, 1990.

[MJ91] M. Meyer and C. Jakfeld. CONTAX: A constraint system for taxonomical knowledge. Research report, German Research Center for AI (DFKI), Kaiserslautern, Germany, 1991. forthcoming.

[MMH85] A. K. Mackworth, J. A. Mulder, and W. S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1:118–126, 1985.

[Mon87] G. Montini. Efficiency considerations on built–in taxonomic reasoning in Prolog. In John McDermott, editor, *Proc. of the 10th International Joint Conference on Artificial Intelligence, Milan*, pages 68–75. Morgan Kaufmann Publishers, Inc., 1987.

[Nud83] B. Nudel. Consistent–labeling problems and their algorithms: Expected–complexity and theory–based heuristics. *Artificial Intelligence*, 21:338–342, 1983.

[Pau90] H.-J. Paulokat. Ein System zur Verarbeitung und Relaxierung von Constraints. Master's thesis, University of Kaiserslautern, Department of Computer Science, November 1990.

[Ric89] M. M. Richter. *Prinzipien der Künstlichen Intelligenz*, pages 126–138. Teubner Verlag, 1989.

[Sap89] M. Sapossnek. Research on constraint–based design systems. In John S. Gero, editor, *Artificial Intelligence in Design - Proceedings of the 4th International Conference on the Applications of Artificial Intelligence in Engineering, Cambridge, UK, July 1989*, pages 385–403. Springer Verlag, 1989.

[Sat90] K. Satoh. Formalizing soft constraints by interpretation ordering. In Luigia Carlucci Aiello, editor, *Proc. of the 9th European Conference on Artificial Intelligence, Stockholm*, pages 585–590. Pitman Publishing, 1990.

[SJ80] G. J. Sussmann and G. L. Steele Jr. CONSTRAINTS – a language for expressing almost–hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[Sri89] N.S. Sridharan, editor. *Proc. of the 11th International Joint Conference on Artificial Intelligence.* Morgan Kaufmann Publishers, Inc., 1989.

[SS89a] M. Schmidt-Schauß. Subsumption in KL–ONE is undecidable. In Brachman et al. [BLR89], pages 421–431.

[SS89b] M. Shanahan and R. Southwick. *Search, Inference and Dependencies in Artificial Intelligence.* Ellis Horwood Series in Artificial Intelligences. Ellis Horwood Limited, 1989.

[Ste90] G.L. Steele Jr. *Common LISP: The Language.* Digital Press, Bedford, MA, 2^{nd} edition, 1990.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[Wal72] D.L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI-TR-271, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1972.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen oder die aktuelle Liste von erhältlichen Publikationen können bezogen werden von der oben angegebenen Adresse.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of currently available publications can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-90-01
Franz Baader: Terminological Cycles in KL-ONE-based Knowledge Representation Languages
33 pages

RR-90-08
Andreas Dengel: A Step Towards Understanding Paper Documents
25 pages

RR-90-02
Hans-Jürgen Bürckert: A Resolution Principle for Clauses with Constraints
25 pages

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 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, and Gert Smolka :
On the Expressivity of Feature Logics with

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

[redacted]

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

RR-91-11
Bernhard Nebel
Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

RR-91-12
J.Mark Gawron, John Nerbonne, and 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-91-15
Bernhard Nebel, Gert Smolka
Attributive Description Formalisms ... and the Rest of the World

RR-91-23
Prof. Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner
Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

RR-91-25
Karin Harbusch, Wolfgang Finkler, Anne Schauder
Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

DFKI Technical Memos

TM-89-01
Susan Holbach-Weber: Connectionist Models and Figurative Speech
27 pages

TM-90-01
Som Bandyopadhyay: Towards an Understanding of Coherence in Multimodal Discourse
18 pages

TM-90-02
Jay C. Weber: The Myth of Domain-Independent Persistence
18 pages

TM-90-03
Franz Baader, Bernhard Hollunder: KRIS: Knowledge Representation and Inference System -System Description-
15 pages

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
A sampler of Relational/Functional Definitions
12 pages

TM-91-05
Jay C. Weber, Andreas Dengel and Rainer Bleisinger
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

DFKI Documents

D-89-01
Michael H. Malburg, Rainer Bleisinger:
HYPERBIS: ein betriebliches Hypermedia-Informationssystem
43 Seiten

D-90-01
DFKI Wissenschaftlich-Technischer Jahresbericht 1989
45 pages

D-90-02
Georg Seul: Logisches Programmieren mit Feature -Typen
107 Seiten

D-90-03
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Abschlußbericht des Arbeitspaketes PROD
36 Seiten

D-90-04
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: STEP: Überblick über eine zukünftige Schnittstelle zum Produktdatenaustausch
69 Seiten

D-90-05
Ansgar Bernardi, Christoph Klauck, Ralf Legleitner: Formalismus zur Repräsentation von Geo-metrie- und Technologieinformationen als Teil eines Wissensbasierten Produktmodells
66 Seiten

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-91-03
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
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 Seiten

HieraCon - A Knowledge Representation System with Typed Hierarchies and Constraints

Bernd Bachmann

D-91-12
Document