Abstraction Techniques for Configuration Systems

Rainer Weigel and **Boi Faltings**
DRAFT
Laboratoire d'Intelligence Artificielle
Ecole Polytechnique Fédérale de Lausanne (EPFL)
IN-Ecublens, CH-1015 Lausanne
Switzerland
{weigel,faltings}@di.epfl.ch

Abstract

In this paper, we address abstraction methods for configuration. Configuration is a design activity where the set of available components and their allowed combinations are known a priori and the goal of the configuration process is to find the sets of components fulfilling the customers wishes and respecting all the compatibility constraints. Abstraction techniques for configuration are becoming more and more important when dealing with the complexity of real world systems. We have developed abstraction methods for finite and discrete constraint satisfaction problems (CSPs) and will describe herein the advantages of using these methods for configuration system. In particular the elaboration of abstraction hierarchies and compacting knowledge using the concept of interchangeability and substitutability on the different levels will be described.

- By using interchangeability we can simply merge objects that are indistinguishable in an abstract space and thereby simplify the problem representation. Other advantages of finding interchangeability and substitutability are that (1) we do not have to bother the user with redundant information on different abstraction levels, and (2) it provides us ways to compact huge amount of data in a way that it can be overlooked by humans and (3) it gives us a possibility for integrating optimization procedures.
- With clustering methods we can transform a constraint based representation on a very fine level of granularity into a more coarse one. This enables us to transform a representation based on parts or components into a representation based on assemblies.

1 Introduction

A simple illustrative example will be given first in order to demonstrate the importance of the concept of *interchangeability* and *substitutability* on the different abstraction levels. In a telecommunication configuration application we will have several analog and ISDN telecommunication facilities. For the ISDN facility a ISDN box and a 220V power supply are necessary. Each ISDN telephone needs a compatible ISDN box which we will call then a ISDN system. These systems might be different in that some can provide additional functionality that other will not provide i.e. with or without a fax for example. However if a customer requires simply a ISDN system without specifying more details, we can see that all the possible systems are interchangeable for such a query. If a customer simply wants a telephone we see that on the next higher level that the analog and the ISDN installations are not interchangeable because the ISDN installations need a power supply. However, if there is a power supply available at the customers place then the analog and the ISDN installations would be interchangeable. This kind of interchangeability is called *context dependent interchangeable* [3]. Similarly if several abstraction levels are con-

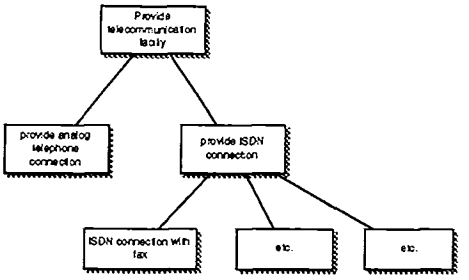

Figure 1: *A function abstraction hierarchy*

sidered at the same time the IDSN installations are substitutable for the analog installation since every functionality the analog system provides can also, beside other, be provided by the ISDN installation. Since we have to assume that a power supply is available we should call it more precisely context dependent substitutability. Thus we can leave the decision about the equipment to come after the choice of the system type (see Fig.: 1). Nevertheless the decision could be force by by decisions on the lower level in the hierarchy.

The following points are important:

- A ISDN telephone can not provide functionality by its own; it must be clustered or combined with a ISDN box. In general, a certain functionality of a product can only be achieved after single parts are clustered into assemblies.
- As soon as a ISDN system fed throught the telephone line is invented we have to recalculate interchangeability. The new systems will be interchangeable respectivly substitutable with analog systems in general and not only in the context where 220V power supply is available. Thus the decision hierarchy must have to be changed ; there is no longer the need of deciding on the type of connection so early in the decision process. Automatically generating such a hierarchy based on the concepts of interchangeability and substitutability is of major importance when changes of the knowledgebase occur frequently.

Abstraction methods have been used successfully in various AI systems [5, 9, 7, 4]. While abstractions were mainly used in those systems to improve search, we will exploit abstractions to facilitate also the interaction with the user . Our abstraction method combines the concept of interchangeability and substitutability with clustering methods.

The goal of this paper is to demonstrate that 1.) configuration problems should be represented on different levels of abstraction and 2.) that the exploitation of interchangeability is in particularly useful for configuration problems. We furthermore claim that due to the internal structure of many configuration problems one can often avoid problems concerning the combinatorics that often appears in random CSPs. We present in the next sections a abstraction technique which is based on *variable abstraction* and *value abstraction.* Variable abstraction clusters variables into a new meta-variable and results after adjusting the constraints in a new CSP which is called *meta-CSP* in [1]. Similarly value abstraction replaces *interchangeable* values into a single meta-value. The concept of *k-interchangeability* [1] plays a major role in the value abstraction procedure and can be seen as a basis for algorithms to calculate partial-interchangeability or meta-interchangeability also mentioned both in [1]. The complete abstraction procedure can be described by applying recursively variable- and value- abstraction procedures until a certain abstraction level is reached. Lets first define the CSPs and describe then the two abstraction methods.

A binary CSP is defined by $P = (X, D, C, R)$, where $X = \{X_1, \ldots, X_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ a set of finite domains associated with the variables, $C = \{C_1, \ldots, C_m\}$ a set of constraints, and $R = \{R_{ij} \subseteq D_i \times D_j$ for C_{ij} applicable to X_i and $X_j\}$ a set of relations that determine the definition of the constraints. Solving a discrete CSP amounts to finding value assignments to variables subject to constraints. The constraint graph of a CSP is a graph $G = (V, E)$ where $V = \{X_1, \ldots, X_n\}$ and $E = \{e_1, \ldots, e_m\}$ with e_i is the edge between the nodes related by C_i.

2 Variable Abstraction

Variables, inducing subproblems of the CSP, are clustered together and treated as a single meta-variable. Each solution to a subproblem becomes a structured value of this new meta-variable. Adjusting the constraints must in turn be done for all the values of each new variable. In configuration problems components can be put together to establish assembly variables.

These new assembly variables itself might be subassemblies of even bigger assemblies and so on. By applying the clustering algorithm several times we will generate what we will call the "abstract graph" of a CSP. For configuration problems, it might turn out that the nodes on the different abstraction levels will also appear in the structural decomposition of the product. Fig 1 shows four constraint graphs of the single constraint problem on different levels of abstraction.

Constraint graphs of problems that arise from physical interactions often resemble clustered graphs. This is in particular true for constraint graphs of product configuration problems with their modular and hierarchical structure. In those graphs we can find high interactions between parts within a assembly and weaker interactions in between different assemblies. These structures are said to exhibit a ultrametric topology [6, 8].

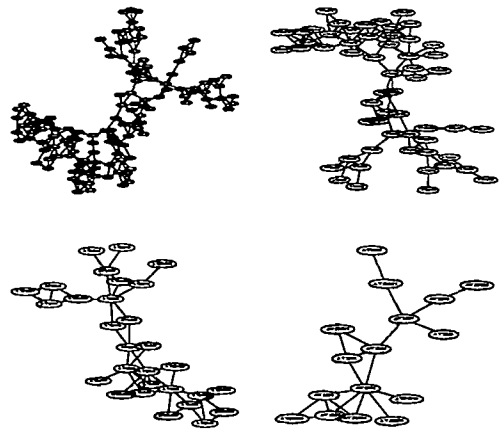

Table 1: Building the abstract graph

In the following we show how to transform a representation on a very fine level of granularity into more abstract representations. In the context of product configuration one would use the structural decomposition of a product as "background knowledge" and run the transformation interactively with the experts. In the example in Fig.: 1 a clustering procedure based on a simplified greedy clique decomposition algorithm was used. Before we present the details of the algorithm to build the abstract graph, we have to describe the composition graph:

Composition Graphs: Let H_0 be a graph with n vertices $v_1, v_2, ..., v_n$ and let

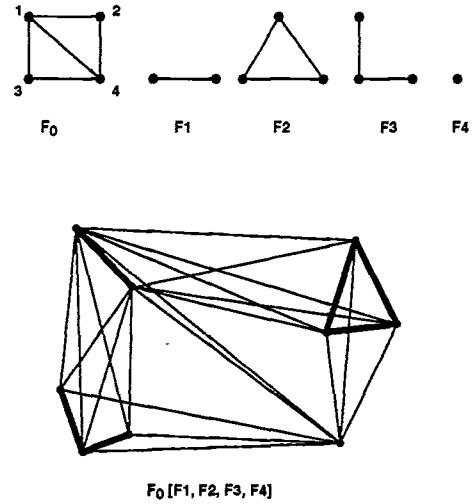

Figure 2: *Example of a composition graph*

$H_1, H_2, ..., H_n$ be n disjoint graphs. The *composition* graph [2] $H = H_0[H_1, H_2, ..., H_n]$ is formed as follows: For all $1 \leq i, j \leq n$, replace vertex v_i in H_0 with the graph H_i and make each vertex of H_i adjacent to each vertex H_j whenever v_i is adjacent to v_j in H_0. H_0 is called the *outer factor and* $H_1, H_2, ..., H_n$ *the* inner factors. Formally, for $H_i = (V_i, E_i)$ we define $H = (V, E)$ as follows:

$$V = \bigcup_{i \geq 1} V_i|$$

$$E = \bigcup_{i \geq 1} E_i \cup \{xy|\ x \in V_i, y \in V_j \text{ and } v_i\ v_j \text{ in } E_0\}.$$

Fig.: 2 shows a composition graph. In the definition above the composition graph is composed given the inner factors and the outer factor. For our abstraction procedure however we have to go the other way round. Given the inner factors $H_1, H_2, ..., H_n$ and the graph G we have to construct H_0. However, we do not require that each vertex of H_i is adjacent to each vertex H_j whenever v_i is adjacent to v_j in H_0. This relaxation simply means that if we have two assembly variables that are related, we do expect that only some components are pairwise connected by a constraint but not all [1]. The inner factor $H_1, H_2, ..., H_n$ necessary for the abstraction procedure are determined by a simplified greedy clique decomposition algorithm. A greedy clique decomposition of a graph G is a partition

[1] Simply adding the universal relation between non-related components of two assemblies makes the procedure conceptually consistent.

of the edges of G into cliques. The size of the decomposition is the number of cliques, and the order of a clique is the number of vertices it contains. Instead of removing the edges from the graph after having found a maximal clique, we remove the nodes contained in the clique; thus we cannot get duplicated nodes. The result of the procedure are a set of *complete* subgraphs of G which build the clusters $H_1, H_2, ..., H_n$. Since we always get complete graphs as clusters and since the graph G is a clustered graph, we expect that only nodes which are closely related [2] are clustered together.

The Abstract Graph: The abstract graph is the result of applying iteratively a "find-cluster" procedure and a abstraction procedure "abstract" as described by the pseudo-code below. Given a graph G the "find-cluster" determines the set of clusters nodes which are then contracted to single nodes by the "abstraction" procedure. In Fig.: 1 we have applied the algorithm three times. The graph in the upper left corner is the constraint graph of a randomly generated CSP with clustered constraint graph. The graph in the lower right corner is the abstract graph of the CSP. One can quite easily see how the structure of the overall problem evolves [3]

```
FUNCTION build abstract graph(G(V,E),levels)
  i :=  0;
  F_0 := G(V,E);
  FOR i := 0 TO levels DO
   H_1, H_2,... H_n := find-cluster(F_i);
   F_i+1 := abstraction(F_i, H_1, H_2,...H_n)
  RETURN F_levels;
END
```

3 Value Abstraction

Having described how the various levels in the abstraction hierarchy of a CSP can be generated, we describe now informally, what it means for values to be interchangeable respectively substitutable. For computational details of the algorithms we refer to [1, 3]. After each iteration in the pseudocode above, i.e. for the new CSP on the next higher abstraction level , we use the concepts of "interchangeability" and "substitutability" to merge equivalent values of new variables. Thereby we neglect redundant values and can focus on the essential parts of the problem on this abstraction level.

Interchangeability: Let us assume that the variables on the lowest level of abstraction represent always classes of atomic parts. For example the class *starter* with the values $\{starter1, starter2, ...\}$. The *engine* variable mentioned above is a meta-variable with structured values are $\{engine1, engine2, ...\}$ where $engine1 = ignition1, starter2, battery - typeC, alternator - XY$ and $engine2 = ...$ etc.

Interchangeability of two parts means that all the values of other variables that are compatible with one part are also compatible with the other. Interchangeability can be found on different levels in the abstraction hierarchy[4]. Interchangeable values for a meta-variable will differ in some values of variables on the more detailed detailed level. Consider the values *engine*1 and *engine*2 for the variable *engine* which could be interchangeable. The two engines differ only in the value of the *ignition* variable (*engine*2 uses *ignition*5 instead of *ignition*1)[5].

This is a simple example where two assemblies A and B provide exactly the same functionality, although single parts in A provide different functionality than the parts of B.

Replacing interchangeable values by a single meta-value is very important when dealing with an abstraction hierarchy and with a huge number of valid configurations. Not only that we can neglect the differences between interchangeable values when we are dealing with meta-values, we can also postpone decisions which are not necessary to make at some point in time during a configuration process. Meta-values give us flexibility in choosing any arbitrary value out of the interchangeable set it represents, without affecting the consistency of the configuration. This feature allows us to integrate configuration optimization procedures easily.

Substitutability: Substitutability is a one way interchangeability, meaning that when value a is substitutable for another value b then we can replace b by a in any solution of the

[2] Remeber the high intra-componetn and the low inter-component interaction in clustered graphs.

[3] The layout is done automatically by the gaph drawing software. Therefor it becomes necessary to rotate the figures adequately. The figure in the upper right corner should be rotated by 180 degree.

[4] Although it is unlikely to find them on the lowest level since they would be completely redundant

[5] It is worth to mention that it is not possible to replace *ignition*5 by *ignition*1 in all engines containing either one one of them i.e. *ignition*5 and *ignition*1 are not fully interchangeable.

CSP containing b, but not vice versa. The algorithm for finding substitutable values will help us to find parts which can always be replaced by some other more "flexible" parts i.e. parts that provide more functionality [6]. Since products evolve during its whole life-cycle it is important that substitutability and interchangeability can be recalculated whenever new parts are designed. Cleaning up a product family by removing substitutable parts or assemblies can have major influence in various parts of a company like inventory management, manufacturing, training of the service personal etc.

4 Functions and Abstractions in Product Configuration Systems:

Many authors have described the importance of reasoning with functions in configuration systems. Design can be defined as the moving from a function description to an attribute description. Based on this definition, N. Murtagh described in an internal IMS working paper primary and secondary functions of an artifact:

- Primary attributes relate directly to the required functions of an artifact, e.g., the primary attributes of a car enable it to perform the functions of transporting people and objects. The desire to have an artifact possessing certain primary attributes is what normally motivates the design in the first place.
- Secondary attributes are normally not essential, e.g. the ease of use of a car or optional features like air-conditioning, sunroofs etc.

Given such a classification of the functions and a abstraction hierarchy, we can see, that primary functions appear on higher levels in the hierarchy only, and secondary functions are more apparent on lower levels. Some functionalities will simply disappear when moving to a higher level. Therefor it is possible, that assemblies providing different functionality can be interchangeable on a higher level of abstraction simply because the their difference is neglected on this level. Recall the example from the beginning, where the function "provide telephone communication" could be realized by a ISDN system as well as by an analog system [7].

[6] Components are often designed to provide more functionality than others components

[7] Assuming power supply is available.

Given the customer requirements related to the primary functionality, we first solve the most abstract CSP. Each solution of the configuration problem on this abstract level is a whole class of "generic" products. If such a generic product should be refined one has simply to follow somehow "cascading menus" where each meta value represents its whole equivalence class. One can think of the configuration process as navigating through a globally consistent solution space. Considering the example in table 1, we could find about 800 solutions on the most abstract level, while we have stopped the search for all solutions on the lowest level prematurely after having found about 5000000 solutions. It is important to realize that this approach is different from simply enumeration of all single solutions, instead we enumerate at each abstraction level classes of solutions, represented by meta-values of clustered meta-variables. We expect that the configuration problem to get smaller and smaller the higher we climb in the abstraction hierarchy and thus solving the problem on the highest level should be quite easy.

User Interaction: Moving in a globally consistent solution space can be quite complicated. Conflicting requirements lead to an over-constraint system and therefor to an empty solution space. The effects on decisions, namely that some others options are no longer possible, must be made explicit. Therefor it is necessary that moving up and down in the abstraction hierarchy must be facilitated. It is important to let the user see the details of some of assemblies while keeping the rest on the more abstract level. This can be realized by providing expansion functions for going into more detail and contraction functions to go backwards. Expansion is demonstrated in Fig.: 3 where node 2 from the outer factor of the composition graph of Fig.: 2 is expanded into F_2. Similarly it can be very useful to have node 2 and F_2 visible at the same time. Setting the value of node 2 gives the framework of a assembly that can be refined within F_2.

These functions enable the user to communicate with the system in a very flexible way so that he can integrate his preferences very easily during the exploration of the solution space.

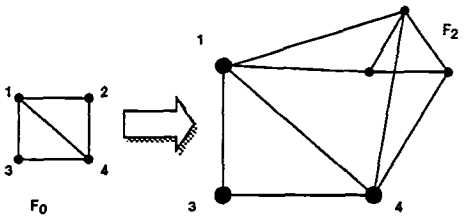

Figure 3: *Node 2 in F_0 expands into F_2*

5 Conclusion

We believe that for many products a salesperson will be able to build by hand a classification scheme of the product variants. However, when he is very deep in the hierarchy he can no longer decide if certain combinations of functionalities are compatible or not. The abstraction hierarchy, we proposed, is build in a bottom up fashion, meaning that we started with a detailed representation and generated a more and more abstract ones. Keeping only the non-redundant information from one level to the next was essential and led to a globally consistent solution space which can be accessed on differnt abstraction levels. The fact, that deciding if certain combinations of some functionalities [8] are valid, can be done in real time within a globally consistent solution space will be extremly important to enable a efficient interaction between user and computer.

With the clustering knowledge provided by a domain expert, we can automatically generate the classification scheme mentioned above. This automation is in particularly important when the product knowledge changes, which can appear very often in the whole product life-cycle, invalidates the schemes.

The algorithms to build the abstraction hierarchy, to solve subproblems, to find interchangeable values etc. will run in a preprocessing or precompilation step. Although all these steps are computationally quite expensive, we believe that it is possible to precompile real world configuration problems in a reasonable amount of time.

References

[1] Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proc. of AAAI-91*, pages 227–233, Anaheim, CA, 1991.

[2] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press Inc., New York, New York, 1980.

[3] R. Weigel, B.V. Faltings, B. Y. Choueiry. Context in discrete Constrint Satisfaction Problems. In *Proc. of the 12 th ECAI*, pages 205–209, Budapest, Hungary, 1996.

[4] E.D. Sacerdoti. *A structure for Plans and Behaviour.* American Elsevier, New York, 1977.

[5] Eearl D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5:115–135, 1974.

[6] H. A. Simon. *The science of the Artificial.* MIT Press, Cambridge,Massachusetts, 1969.

[7] Mark Stefik. Planning with constraints (molgen: Part 1). *Artificial Intelligence*, 16(2):111–140, 1981.

[8] C. P. Wiliams and T. Hogg. Exploiting the deep structure of constrint problems. *Artificial Intelligence*, 70:73–117, 1994.

[9] D. E. Wilkins. Domain-independent planning: representation and plan generation. *Artificial Intelligence*, 22:269ff, 1984.

[8] The functionalities can obviously be expressed on different levels