Configuration as Composite Constraint Satisfaction

Daniel Sabin and **Eugene C. Freuder**
Department of Computer Science
University of New Hampshire
Kingsbury Hall M208
Durham, NH 03824
Phone (603) 862-1867
ds1@cs.unh.edu, ecf@cs.unh.edu

Abstract

Selecting and arranging parts is the core of a configuration task. The validity of a configuration is defined in terms of constraints. Highly declarative, domain independent and simple to use, the *constraint satisfaction problem (CSP)* paradigm offers an adequate framework for this task. However, the basic paradigm is not powerful enough to capture or to take advantage of essential aspects of configuration, such as the unknown a priori number of constituent parts of a system or the inherent internal structure of these parts. Although notable effort has been spent on extending the basic paradigm to accommodate these issues, we still lack a comprehensive formalism for configuration. This paper presents the main ideas behind a general constraint-based model of configuration tasks represented as a new class of nonstandard constraint satisfaction problems, called *composite CSP*. Composite CSP unifies several CSP extensions, providing a more comprehensive and efficient basis for formulating and solving configuration problems.

Introduction

Configuration can be informally defined as a special case of design activity, with the key feature that the artifact being designed is assembled from a fixed set of predefined components that can only be connected in predefined ways. Selecting and arranging combinations of parts which satisfy given specifications is the core of a configuration task. The specification of configuration tasks, in general, involves two distinct phases, the description of the domain knowledge and the specification of the desired product. The domain knowledge describes the objects of the application and the relations among them. The specifications for an actual product describe the requirements that must be satisfied by the product and, possibly, optimizing criteria that should be used to guide the search for a solution. The solution has to produce the list of selected components and, as important, the structure and topology of the product.

Highly declarative, domain independent and simple to use, the *constraint satisfaction problem (CSP)* paradigm offers an adequate framework for this task. Constraint satisfaction problems (CSPs) involve finding values for problem variables subject to constraints that restrict which combinations of values are allowed. Unfortunately, this formalism is not powerful enough to capture or to take advantage of essential aspects of configuration, such as the unknown a priori number of constituent parts of a system or the inherent internal structure of these parts.

The main contribution of this paper is the definition of a new, nonstandard class of CSPs, called *composite constraint satisfaction problems*, which is used to model configuration tasks. A composite CSP is an extension of the classical paradigm, in the sense that *values for variables in a composite CSP can be entire subproblems*. It subsumes three previous CSP extensions: *dynamic CSPs*, *hierarchical domain CSPs* and *meta CSPs*, incorporating both simplification and further extension, offering increased representational power and efficiency. As an example, part of a car configuration problem represented using our formalism is shown in Figure 1.

The composite CSP is presented in the form of a constraint graph, where variables are the vertices (represented as rectangular boxes) and constraints are the edges. If more than one choice is available at one vertex, the alternatives are specified inside the box, in italics. For simplicity, we will assume here that our problem is *binary* (all the constraints involve at most two variables).

According to the specifications, a car is composed of four functional subsystems – the power plant, the power train, the support system and the control system – represented in our formalism by four variables. Each subsystem in turn has its own functional composition, that can be represented as a separate composite CSP. For example, the power plant includes the engine and its fuel, exhaustion, lubrication, cooling and electrical systems. Sometimes it is possible to have several choices in implementing a (sub)system. We represent the alternatives as the set of possible values (the domain) of a variable. Considering only piston engines,

we have two choices for engine, gasoline engine and diesel engine, respectively. We represented only the model for the gasoline engine in the graph, but a similar model exists for the diesel engine. Some systems consist of smaller functional subsystems and they are represented as a variable with a single-value domain, e.g. the electrical system. We should note here that, in order to keep the picture simple, we do not explicitly represent single-value domains in the graph and some systems remain unspecified, e.g. power train, support system, etc.

The constituent parts of a system are subject to different restrictions, imposed by technical, performance, manufacturing and financial considerations, to name just a few. For example, choosing a *TypeA* battery will restrict the list of options to *PackageA* alone, because the other two packages require a much stronger type of battery, *TypeB*. On the other hand, a *TypeB* battery, being more expensive, is unnecessary for *Package A*, which can be handled by a weaker battery. Or, by selecting the *Sport* category, the engine type is restricted accordingly, etc. All these restrictions are captured in our model in the form of constraints, which are relations over sets of variables, usually represented extensionally in the form of sets of allowed tuples of values. The previous constraint, between the options list and battery type is thus represented as $C_{Options,Battery} = \{$ *(PackageA, TypeA), (PackageB, TypeB), (PackageC, TypeB)* $\}$.

The paper is further organized as follows. Section 2 contains some background information about CSPs and previous extensions to the basic paradigm. Section 3 contains an overview of the most important previous approaches to modeling configuration tasks, showing both the contributions they made and their deficiencies. We describe in detail composite CSPs and how they are used for modeling configuration tasks in Section 4. Our future research directions are outlined in Section 5.

CSP Background

Constraint satisfaction problems involve finding values for problem variables subject to restrictions on which combinations of values are allowed (Tsang 1993). A CSP is thus defined as a triplet $\mathcal{P} =< \mathcal{V}, \mathcal{D_V}, \mathcal{C_V} >$, where

(1) $\mathcal{V} = \{V_1, \ldots, V_n\}$ is the set of *variables*,

(2) $\mathcal{D_V} = \{dom_{V_i} \mid V_i \in \mathcal{V}\}$, is the set of *domains*, with $dom_{V_i} = \{val_{i_1}, \ldots, val_{i_p}\}$ representing the set of all possible values for variable V_i, and

(3) $\mathcal{C_V} = \{C_S \mid S \subseteq \mathcal{V}, S \neq \phi\}$ is the set of *constraints*. A constraint C_S is a relation over the variables in S and can be represented extensionally as a subset of the Cartesian Product of the domains of those variables.

If the arity of all the constraints in $\mathcal{C}$ is at most 2, we have *binary* CSPs, as opposed to *general* (or n-ary) CSPs otherwise. Binary CSPs are usually represented in the form of *constraint graphs*, where vertices in the graph correspond to variables and edges correspond to constraints, as presented in Figure 1.

The goal in solving a CSP is to assign a value to

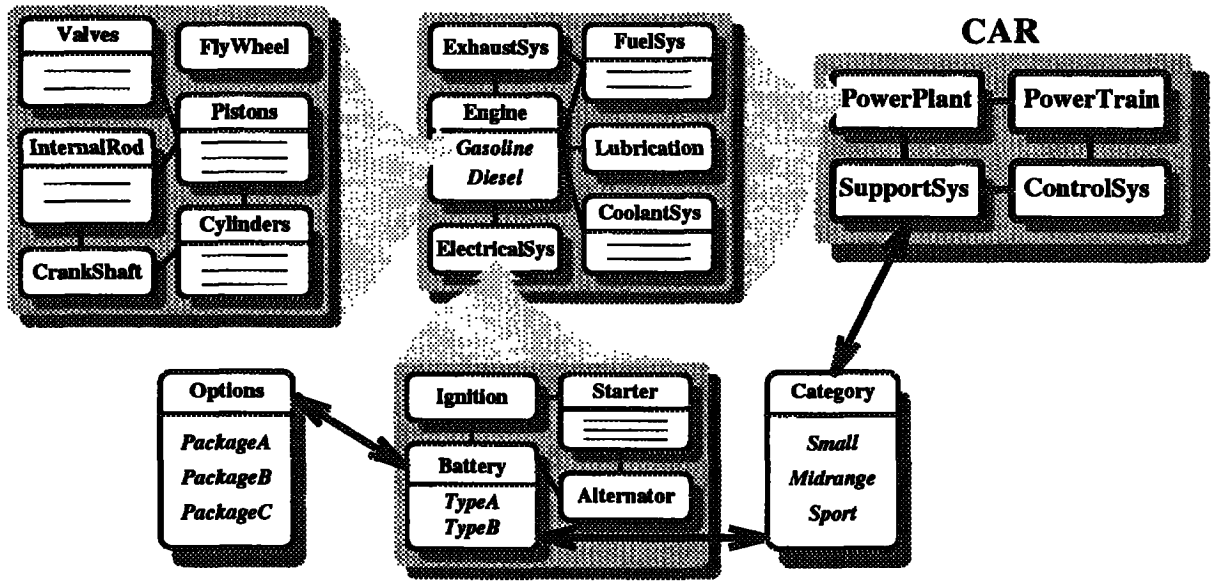

Figure 1: Sample configuration problem represented as a composite CSP

each variable, such that all the constraints are simultaneously satisfied. Constraint satisfaction is NP-complete in general. The main algorithm for solving a CSP is *backtrack search*. A standard measure of the search effort is represented by the number of *constraint checks* the algorithm performs. To improve efficiency, consistency inference (constraint propagation) techniques are used to prune values before or during search. The basic pruning technique involves establishing or restoring some form of *arc consistency*. Given a constraint C_{V_i,V_j}, if a value val_{V_i} for variable V_i is not consistent with any value for variable V_j, than val_{V_i} is *arc inconsistent* and can be removed. *Full arc consistency* is achieved when all arc inconsistent values are removed.

Hierarchical Domain Constraint Satisfaction Problems

In many applications, in general, and in engineering applications, in particular, the domain values cluster into sets with common properties and relations. This categorization can be represented as a specialization/generalization (IS-A) hierarchy (Havens & Mackworth 1983). If the domains of all the variables are organized hierarchically, we have a hierarchical domain CSP. Several arc-consistency algorithms, which use this property, have been proposed in the literature (Mackworth, Mulder, & Havens 1985) (Kokeny 1994). Although they are based on different CSP arc consistency algorithms, they all use the same idea. For every variable, the domain of values contains all the nodes (internal, as well as leaves) in the corresponding hierarchy. A breadth-first linearization of the directed-acyclic graphs representing the hierarchical domains is used to impose a partial order on these flattened domains. Two inferences can be made about domain values:

(1) if a value high in the hierarchy is found to be arc inconsistent, then so are all its descendents, and

(2) if a value lower in the ordering is found to be arc consistent, then so are all its ancestors.

and they allow the algorithm to avoid a potentially large number of constraint checks. Making the stronger assumption that the hierarchies preserve the compatibility (that is, if some value of a variable satisfies a constraint, than any specialization of that value also satisfies the constraint), a more efficient algorithm is presented in (Kokeny 1994).

Dynamic Constraint Satisfaction Problems

According to the CSP definition, the set of variables and the set of constraints have to be exactly specified, as part of the problem. On the other hand, in many reasoning tasks, especially in synthesis tasks, the set of variables that participate in a solution cannot be determined *a priori*. To cope with this problem, (Mittal & Falkenhainer 1990) introduces the notion of *dynamic constraint satisfaction problem (DCSP)*. The main idea is that only a subset of all the variables need to be instantiated and, thus, part of a solution. Each variable can be in one of two states: *active* or *inactive*. Furthermore, the constraints are divided into two specialized types: *Require Variable (activity)* constraints, that specify conditions on the values of already active variables, under which new variables become active, and *Compatibility* constraints, which specify relations on consistent values for variables. Inferences can now be made about the status of a variable, based on the activity constraints, avoiding irrelevant work during search.

A DCSP is described by the set of all variables that may potentially become active, a nonempty initial set of active variables, the set of domains and the two sets of activity and compatibility constraints. The search process starts with the initial set of active variables. Additional variables are introduced as the result of satisfying activity constraints. The problem is thus changing dynamically as the search progresses. The set of active variables induces the set of "active" constraints, in the sense that a constraint which does not have all its variables active is by definition trivially satisfied, and thus, becomes "inactive".

A formal, mathematically well-founded treatment of the conditional existence of variables is presented in (Bowen & Bahler 1991) where the constraint network is viewed as a set of sentences in first-order free logic.

Meta Constraint Satisfaction Problems

In the CSP context, when we talk about structure in general, we can position ourselves on one of three levels: *macro, micro* and *meta* (Freuder 1992). At the meta level, we decompose the problem into subproblems, and view this decomposition as a *metaproblem*.

Each *metavariable* of the metaproblem is a subproblem of the original problem. The domain of a metavariable is the set of solutions to the subproblem. Each subproblem includes a subset of the variables in the original problem, together with the values for these variables and the constraints relating variables within this subset. Metavariables can overlap by sharing common variables. A *metaconstraint* between two metavariables must enforce all the original constraints, involving variables from the corresponding subproblems. Furthermore, if the same variable appears in both subproblems, the metaconstraint must ensure that this variable receives the same value in the solution chosen as *metavalue* for each of them.

The goal of this decomposition is to deal with the complexity of solving a problem by solving an equivalent problem, represented at a different level of abstraction.

This is desirable because either the metaproblem or the metavariable subproblems may present a special structure, which can be solved more efficiently.

Configuration Paradigms

An important contribution towards a generic domain-independent model of configuration tasks was presented by Mittal in (Mittal & Frayman 1989) (Mittal & Falkenhainer 1990). He makes the observation that two important assumptions hold for most of the configuration tasks:

(a) artifacts are configured according to some known *functional architectures*, each abstractly defined by *functions*, and

(b) for each function, there is some *key component* (or a small set) that is crucial in implementing the function.

According to a recent survey of configuration systems (Stefik 1995), configuration problem solving methods work through well defined phases. Based on the above assumptions, user specifications lead to an abstract configuration, where goals are represented in terms of the desired functions the system has to provide, which in turn translates to the set of (abstract) key components required to implement these functions. This abstract solution then undergoes an expansion and refinement process until a complete, detailed configuration is obtained. Different bodies of domain specific knowledge guide this process.

Several approaches have been proposed for solving configuration tasks. They differ in the kinds of knowledge they take into consideration and how is this knowledge organized, and in the methods they propose for solving the problem.

Although most of the implemented configuration systems to date use *production rules* as the mechanism for representing configuration tasks (McDermott 1982) (Birmingham *et al.* 1988), this approach is not well suited for large and complex problem domains, with a high rate of change of the knowledge. The well known drawback of rule-based systems is the lack of separation between control and domain knowledge, which makes knowledge maintenance a very difficult task. To get an idea about the complexity of this task, XCON, a well known knowledge-system for configuring computer systems at Digital Equipment Corporation, had in 1989 in its knowledge base more than 31,000 components and approximately 17,500 rules (Barker & O'Connor 1989). The change rate of this knowledge is about 40 percent per year. This challenge lead to the development of a programming methodology that provides structuring concepts for rule-based programming (Bachant 1988). Meta rules are used to control and order context specific decisions, but practice shows that the maintenance problem is still unsolved.

As the name shows, the *resource-based* approach (Heinrich & Jungst 1991) is centered around the notion of resource, which is a kind of service provided or consumed by a component. It offers a producer-consumer model of a configuration task, in which each entity in the model knows about the amount of resources it produces, consumes and uses. The goal is to find a set of components that bring the overall set of resources in a balanced state, where all the demands are fulfilled.

The inference scheme proposed, guided by the requirements which are still unsatisfied, proves to be efficient and adequate for configuration tasks where the goal is to provide certain amounts of functionality only. The algorithm produces a list of components that cover the range of intended functionality, but in configuration in general it is as important to specify, as part of the solution, the structure and the topology of the desired artifact. Taking into account only the functional dependency relation among components, the resource-based approach cannot reason about structure. A large class of configuration problems, where the main concern is how to obtain a valid configuration, which satisfies complex restrictions on how components can be interconnected, cannot be represented and solved using this framework. Although further extensions in this direction are possible, there are still important open questions, on the termination conditions, soundness and completeness of the proposed inference scheme, that have not been answered yet.

A special characteristic of configuration tasks is that the problem is well structured and completely specified: the description of all the components is complete and all the relations and constraints among different parts are stated explicitly. Most of the complexity of solving a configuration problem lies in the domain knowledge, not in the method itself. The difficulties to be addressed are knowledge representation, efficient knowledge application and mechanisms for coping with the high rate of change of the knowledge base. The CSP paradigm offers, as we mentioned earlier, an adequate framework for addressing these issues. However, the basic paradigm fails to capture the dynamic and hierarchical nature of the domain knowledge. Although notable effort has been spent on extending the basic CSP paradigm to overcome these limitations (dynamic CSPs, hierarchical domain CSPs), we still lack a comprehensive formalism for configuration.

In the model presented by Mittal (Mittal & Falkenhainer 1990), configuration tasks are represented as dynamic CSPs. In order to deal with entities that can be treated as resources, two new types of variables are introduced in a companion paper, *pool* and *cumulative* variables. Furthermore, methods are described for reasoning with a wide range of resources, from the simple case of a single atomically sharable component to the more general case of differentiated and replicated resources.

This extended framework is considerable more powerful, but it still does not offer a comprehensive model of configuration tasks. The component structure demanded in the general configuration framework remains unclear, the approach does not take into consideration the hierarchical nature of the domain knowledge and does not offer support for reasoning on sets

of configuration entities, where the elements of these sets are not known a *priori*.

In the constraint-based Configuration framework ConfCS presented by Haselbock (Haselbock 1993), configuration tasks are represented as DCSPs using the same classes of constraints as Mittal, that is, Require-Variable and Compatibility constraints. Constraints are specified on a meta level, as relations among component types, which allows the treatment of multiple occurrences of components with similar behavior, but there is no support for reasoning about shared vs. exclusive requests, atomic vs. partial use of components, etc. Another limitation of this approach is that, like Mittal's approach, it does not use the hierarchical nature of the domain knowledge.

Composite Constraint Satisfaction Problems

In this paper we propose a different constrained-based approach to modeling configuration tasks based on a new, nonstandard, class of CSPs called *composite CSPs*.

Definition

A composite CSP is defined in the same manner as classical CSPs, by giving the triplet $< \mathcal{V}, \mathcal{D}_\mathcal{V}, \mathcal{C}_\mathcal{V} >$. The difference consists in the fact that the values a variable can take are not anymore restricted to be atomic values, as in the classical case. Instead, a value can be an entire *subproblem* $\mathcal{P}' =< \mathcal{V}', \mathcal{D}_{\mathcal{V}'}, \mathcal{C}_{\mathcal{V}'} >$. The effect of instantiating a variable U to the value $\mathcal{P}'$ is that the problem we have to solve is dynamically modified and becomes $\mathcal{P}'' =< \mathcal{V}'', \mathcal{D}_{\mathcal{V}''}, \mathcal{C}_{\mathcal{V}''} >$, where

- $\mathcal{V}'' = \mathcal{V} \cup \mathcal{V}' - \{U\}$,
- $\mathcal{D}_{\mathcal{V}''} = \mathcal{D}_{\mathcal{V}'} \cup \{dom_X \mid X \in \mathcal{V}, X \neq U\}$, and
- $\mathcal{C}_{\mathcal{V}''} = \mathcal{C}_\mathcal{V} \cup \mathcal{C}_{\mathcal{V}'} \cup \mathcal{C}_{\mathcal{V}',\mathcal{V}} - \mathcal{C}_{\{U\}}$, where
  $\mathcal{C}_{\{U\}}$ is the set of all constraints between variable U and other variables in $\mathcal{V}$, and $\mathcal{C}_{\mathcal{V}',\mathcal{V}}$ is the set of all constraints connecting a variable in $\mathcal{V}'$ to a variable in $\mathcal{V}$. The constraints in $\mathcal{C}_{\mathcal{V}',\mathcal{V}}$ can be seen as a refinement of the constraints in $\mathcal{C}_{\{U\}}$ in the sense that previous restrictions imposed on U, by its neighbors in the constraint graph, become now more precise, referring explicitly to variables in U's internal structure.

One of the most important advantages of our approach is that any previous CSP algorithm can be adapted in a straightforward manner to solve a composite CSP. As the result of instantiating a variable v to value *val*, $\mathcal{V}$, $\mathcal{D}$ and $\mathcal{C}$ change dynamically, according to the definition. In case this instantiation does not lead to a solution, these changes have to be undone. Since no other modifications are necessary, all existing filtering and search algorithms can thus be easily adapted and used, without the need for introducing specialized, conceptually different methods.

Comparison Between Composite CSP and Previous CSP Extensions

We will show further how several previous CSP extensions, presented earlier in Section 2, are subsumed by the notion of composite CSP, pointing out the advantages of our approach. Given the space limitations, we concentrate on the case of hierarchical domain CSPs, and give a very brief description for the other two.

Meta CSPs First, it might seem that the expansion process implied by the instantiation of a variable in a composite CSP is exactly the opposite of the abstraction process that takes place in meta CSPs, when a subproblem is replaced by a meta variable. In some sense this is true, but the difference is more subtle than that. In a meta CSP a metavariable is associated with a fixed, static subproblem. In a composite CSP, several subproblems can be plugged in at any variable, through a dynamic process, the decision being based on the current state of the problem, represented by the set of values at other variables.

A problem solving method that is rapidly gaining acceptance in many application areas is *model synthesis*. The goal of this method is to construct or modify a model of a system in accordance with the domain knowledge. From this perspective, the configuration task can be viewed as the task of constructing a detailed model of a system in accordance with a set of specifications. Similarly, diagnosis can be viewed as the task of altering an existing model, describing the correct behavior of the system under diagnosis, to make it comply with a set of observations which reflect the system's actual behavior.

In a composite CSP, when we instantiate a variable to a set value, we dynamically change the problem at hand. We can accomplish two different goals by doing this. First, we can extend the model, by refining parts of it. Second, when we select one of the values available in the domain of some variable, we chose in fact among alternate models for the same (sub)system, each one being adequate for a specific task.

Hierarchical Domain CSPs The optimal worst case space complexity for representing a binary constraint given in its extensional representation and involving two variables with a domain size of d is $O(d^2)$. Since typical real applications involve thousands of constraints and hundreds of values for each variable, at this space complexity we cannot even represent the entire problem. One of the reasons for introducing the notion of composite CSPs was the need for an efficient mechanism to deal with this issue in the case of CSPs with hierarchical domains. The dynamic nature of the composite CSP approach allows us to cut down the space complexity by considering only a small part of the problem at any given time. Consider the following small example representing part of the hypothetical car configuration problem introduced in Figure 1. We have the two variables, *Category* and *Engine* and one con-

straint, $C_{Category,Engine}$, given explicitly in the form of the set of allowed tuples:

$C_{Category,Engine} = \{$
$(Compact, 2.0L4)$, $(Compact, 2.2L4)$,
$(Compact, 2.5L6)$, $(Compact, 4D)$,
$(MidSize, 2.0L4)$, $(MidSize, 2.2L4)$,
$(MidSize, 2.5L6)$, $(MidSize, 3.0L6)$,
$(MidSize, 3.2V6)$, $(MidSize, 4D)$,
$(MidSize, 4TD)$, $(FullSize, 2.0L4)$,
$(FullSize, 2.2L4)$, $(FullSize, 2.5L6)$,
$(FullSize, 3.0L6)$, $(FullSize, 3.2V6)$,
$(FullSize, 4.6L8)$, $(FullSize, 4D)$,
$(FullSize, 4TD)$, $(Coupe, 3.2V6)$,
$(Coupe, 4.6L8)$, $(Sedan, 5.7L8)$
$\}$

The domains of values for the two variables are hierarchically organized, as shown in Figure 2.

To represent this problem as a composite CSP, we do the following:

- ▷ we eliminate from all the hierarchies the internal nodes which have only one descendant. For example, we eliminate node *Small* from the hierarchy presented in Figure 2.
- ▷ we associate a variable with each internal node that remains in the hierarchy. We thus have variables *Category, Midrange, Sport, Engine, Diesel, Gasoline, 4-Cyl, 6-Cyl* and *8-Cyl*.
- ▷ The domain of possible values for each variable is represented by the set of its direct descendants:

  $dom_{Category} = \{ Compact, Midrange, Sport \}$
  $dom_{Midrange} = \{ MidSize, FullSize \}$
  $dom_{Sport} = \{ Coupe, Sedan \}$
  $dom_{Engine} = \{ Diesel, Gasoline \}$
  $dom_{Diesel} = \{ 4D, 4TD \}$
  $dom_{Gasoline} = \{ 4-Cyl, 6-Cyl, 8-Cyl \}$
  $dom_{4-Cyl} = \{ 2.0L4, 2.2L4 \}$
  $dom_{6-Cyl} = \{ 2.5L6, 3.0L6, 3.2V6 \}$
  $dom_{8-Cyl} = \{ 4.6L8, 5.7L8 \}$

- ▷ Consider, in the general case, a constraint $C_{X,Y} \subset dom_X \times dom_Y$. If it contains at least one pair of values (x_i, y_j) such that either x_i or y_j has predecessors, other than the root, in its hierarchy, then the constraint is replaced by a set of *implied constraints*. We define an implied constraint $C_{U,V}$ for every pair of nodes U and V, where U is a predecessor of some value in dom_X and V is a predecessor of some value in dom_Y. For each pair of values (u, v) allowed by $C_{U,V}$, at least one of the following has to be true:
  - (a) $(u, v) \in C_{X,Y}$
  - (b) $u \in dom_X$ and there exists a descendant y of v s.t. $y \in dom_Y$ and $(u, y) \in C_{X,Y}$
  - (c) $v \in dom_y$ and there exists a descendant x of u s.t. $x \in dom_X$ and $(x, v) \in C_{X,Y}$
  - (d) there exists a descendant x of u and there exists a descendant y of v s.t. $x \in dom_X$ and $y \in dom_Y$ and $(u, y) \in C_{X,Y}$

  NOTE. If an implied constraint is trivial, that is, it allows all the possible combinations of values or it does not allow any combination of values at all, then we eliminate it.

The resulting representation is presented in Figure 3. Bellow are the implied constraints for our example, expressed as sets of allowed pairs of values.

$C_{Engine,Sport} = \{$
$(Gasoline, Coupe)$, $(Gasoline, Sedan)$
$\}$
$C_{Diesel,Category} = \{$
$(4D, Compact)$, $(4D, Midrange)$ $(4TD, Midrange)$
$\}$
$C_{Gasoline,Category} = \{$
$(4-Cyl, Compact)$, $(4-Cyl, Midrange)$,
$(6-Cyl, Compact)$, $(6-Cyl, Midrange)$,
$(6-Cyl, Sport)$, $(8-Cyl, Midrange)$,
$(8-Cyl, Sport)$
$\}$
$C_{Gasoline,Midrange} = \{$
$(4-Cyl, MidSize)$, $(4-Cyl, FullSize)$,
$(6-Cyl, MidSize)$, $(6-Cyl, FullSize)$,
$(8-Cyl, FullSize)$
$\}$
$C_{Gasoline,Sport} = \{$
$(6-Cyl, Coupe)$, $(8-Cyl, Coupe)$,
$(8-Cyl, Sedan)$
$\}$

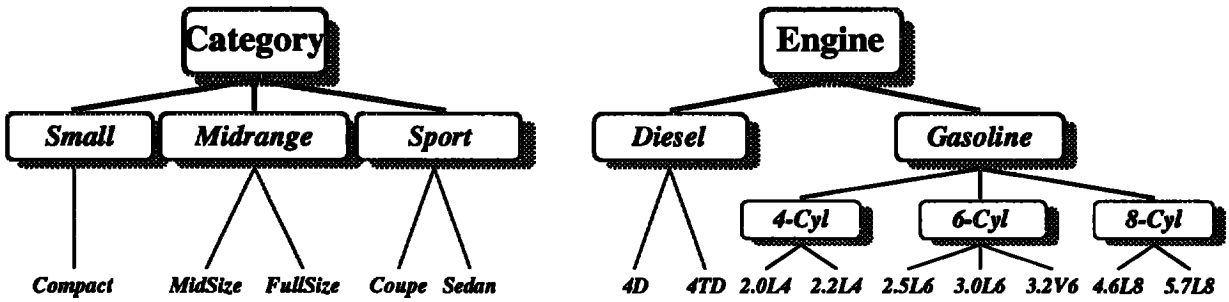

Figure 2: Sample hierarchical organization of domains of values

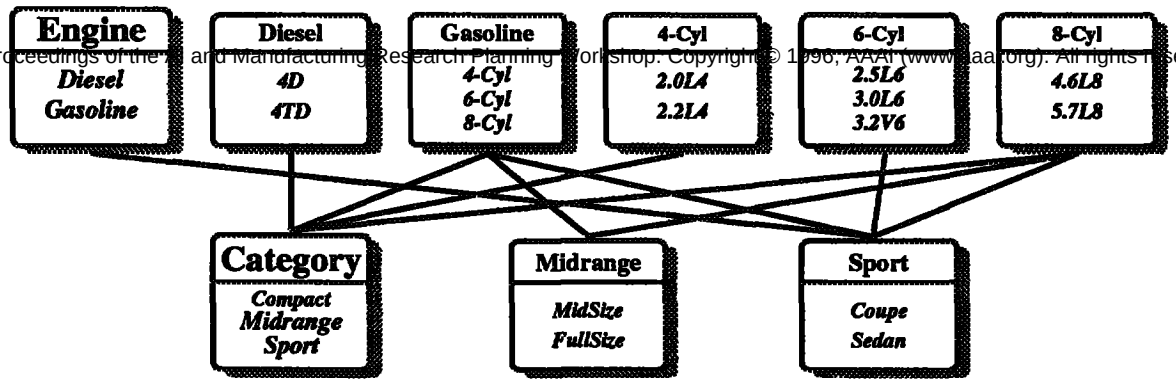

Figure 3: CSP with hierarchical domains represented as composite CSP

$C_{4-Cyl,Category} = \{$
$(2.0L4, Compact), \quad (2.0L4, Midrange),$
$(2.2L4, Compact), \quad (2.2L4, Midrange)$
$\}$
$C_{6-Cyl,Sport} = \{ \quad (3.2V6, Coupe) \quad \}$
$C_{8-Cyl,Category} = \{$
$(4.6L8, Midrange), \quad (4.6L8, Sport),$
$(5.7L8, Midrange), \quad (5.7L8, Sport)$
$\}$
$C_{8-Cyl,Midrange} = \{ \quad (4.6L8, FullSize) \quad \}$
$C_{8-Cyl,Sport} = \{ \quad (4.6L8, Coupe) \quad (5.7L8, Sedan) \quad \}$

It seems that instead of offering a simpler representation, with a smaller size complexity, we ended up with a much larger problem, 9 variables and 10 constraints, compared to the 2 variables and 1 constraint required by the original problem. In fact, at any moment during search, the composite CSP consists of only 2 variables and at most 1 constraint. As shown in Figure 4(a), initially the problem is defined by the variables *Engine* and *Category*, and no constraint between them. By choosing value *Gasoline* for *Engine*, we change the problem to the one in Figure 4(b), defined by the variables *Gasoline* and *Category*, and the constraint $C_{Gasoline,Category}$. Selecting further *Midrange* for *Category* and $8-Cyl$ for *Gasoline*, the problem moves in the state presented in Figure 4, and a solution to the problem is $(4.6L8, FullSize)$.

Furthermore, if you recall from the definition of hierarchical domain CSPs, the domain of a variable consists of all nodes in the hierarchy and, thus, can be very large. Each variable in our representation has a much smaller domain, consisting of the set of its direct descendants in the hierarchy. For the example above, the hierarchical domain CSP representation would have $| dom_{Category} | = 9$ and $| dom_{Engine} | = 15$, while the maximum domain size for any variable in the composite CSP representation is 3.

In conclusion, we obtain a composite CSP having the same number of variables as the original CSP or as the hierarchical domain CSP, at most the same number of constraints and much smaller domains, thus reducing the space complexity.

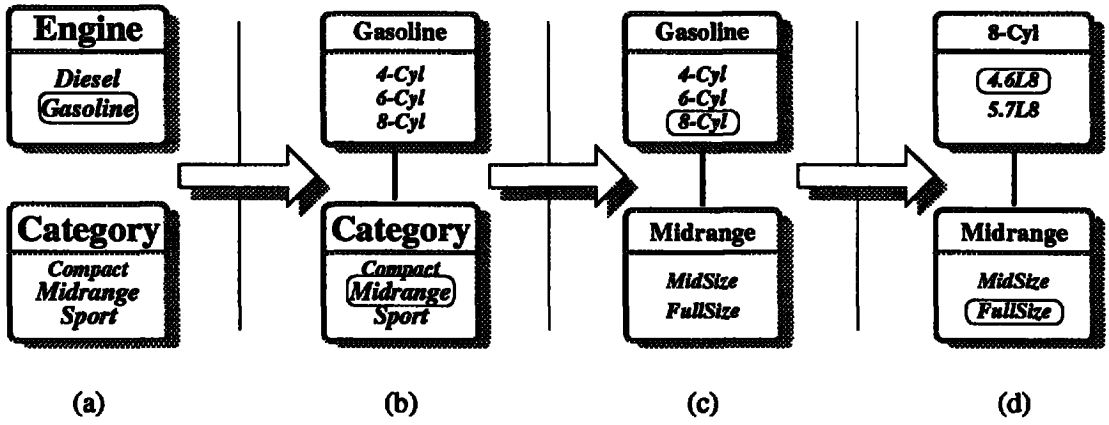

Figure 4: Solving a Composite CSP

Dynamic CSPs An important aspect of modeling, as a human activity, is the level of detail to which the system is described. The definition of a composite CSP introduces a dual viewpoint on the objects involved in a problem. At different levels of detail, the same object can be viewed both as a value and as a subproblem, as shown in the example presented at the beginning of this paper in Figure 1. A constraint in our approach thus plays a dual role. At a higher level of detail, the constraint can be viewed as what in dynamic CSP terms is called a compatibility constraint, specifying restrictions on what combinations of values are allowed for a set of variables. At a lower level of detail, the same constraint can be seen as an activity constraint, in the sense that it specifies under which conditions new variables are created and become part of the problem.

For example, if variable *Engine* is successfully instantiated, according to the restrictions imposed by the values at the neighboring variables, to value *Gasoline*, variables *Valves*, *InternalRod*, *FlyWheel*, *Pistons*, *Cylinders* and *CrankShaft* are created and added to the problem.

Future Research Topics

We are currently investigating a formalism which will allow the user to specify a configuration problem by describing the domain knowledge and the requirements for the desired product, including any optimization criteria that should guide the search for a solution. What we have in mind, at this point, is to develop a very simple knowledge representation language, which combines the simplicity of the CSP paradigm with the power of the object-orientation paradigm. The reasons that motivate our decision are the following.

There are two main sources of domain knowledge in configuration tasks. The first one refers to *component description*. Configuration reasons under a *closed world* assumption, that is, the domain description is complete and covers the space of all possible solutions. For any given configuration domain, whether we are successful or not in solving a configuration task depends on the success of designing configurable components that can be interconnected systematically, covering the entire range of possible functions. The ability of identifying a commonality of characteristics is thus crucial for modeling configuration. On the other hand, organizing these characteristics in hierarchies is an important aspect of object-oriented modeling. In the process of grouping together similar objects in classes, common structure and functionality are factored out, while specific aspects are identified.

Another central theme of object-oriented modeling is *abstraction*. In general, abstraction is a mechanism for coping with complexity. It offers a way of concentrating only on what is important for solving a specific problem. In object-oriented modeling, it allows us to decide which characteristics of an object are important for the model, and which ones are unimportant. As we have already emphasized, the set of component types available in a configuration domain is finite. Each part, viewed as a component type, is usually a complex artifact that can be defined, in a declarative manner, by the set of its essential characteristics which, within the application domain, make it different from other parts. There is a clear distinction between properties that define descriptive features of the object and properties that define the relation with other objects. One of the reasons for the success of object-oriented techniques is their ability to capture the essence of both structure and behavior in the single abstraction of an *object*. The main advantage in doing this is that parts can be described individually, independent of the ways in which they will be used.

The second source of domain knowledge is the *compatibility and dependency information*. Components can be interconnected only in predefined ways and can play different roles with respect to each other. All this knowledge can be expressed very simple, in a declarative manner, as constraints among component types and their properties. The optimization criteria can be expressed in a similar way.

As we mentioned earlier, we assume that the functional composition of the system to be configured is known. This means that, apart for the optimization criteria, the system can be viewed, recursively, as a larger component, and thus described, as an object, aggregated from a set of parts. Of course, the description will probably contain abstract parts and will not be complete, in the sense that the existing parts might require additional parts in order to function properly. But this is the task of the configuration system: starting with a (partial) abstract configuration, to find a complete and irreducible configuration implementing the given functional composition.

Once the problem is expressed using the representation language, as described above, the translation into the composite CSP representation is direct.

Acknowledgements This material is based on work supported by a grant from ILOG SA.

References

Bachant, J. 1988. Rime: Preliminary work toward a knowledge-acquisition tool. In Marcus, S., ed., *Automating Knowledge Acquisition for Expert Systems.* Boston, MA: Kluwer Academic Publisher. 201–224.

Barker, V., and O'Connor, D. 1989. Expert system for configuration at digital: Xcon and beyond. *Communications of the ACM* 32(3):298–318.

Birmingham, W.; Brennan, A.; Gupta, A.; and Sieworek, D. 1988. Micon: A single board computer synthesis tool. *IEEE Circuits and Devices*:37–46.

Bowen, J., and Bahler, D. 1991. Conditional existence of variables in generalized constraint networks. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 215–220.

Freuder, E. 1992. Constraint solving techniques. In Mayoh, B., Tyugu, E., and Penjaen, J., eds., *Constraint Programming*, volume 131 of *Series F: Computer and Systems Sciences*. NATO ASI Series. 51–74.

Haselbock, A. 1993. *Knowledge-based Configuration and Advanced Constraint Technologies*. Ph.D. Dissertation, Institut fur Informationssysteme, Technische Universitat Wien.

Havens, W., and Mackworth, A. 1983. Representing knowledge of the visual world. *IEEE Computer* 16(10):90–96.

Heinrich, M., and Jungst, E. 1991. A resource-based paradigm for the configuring of technical systems from modular components. In *Proceedings of the Seventh IEEE Conference on Artificial Intelligence Applications*, 257–264.

Kokeny, T. 1994. A new arc-consistency algorithm for csps with hierarchical domains. In *Workshop Notes of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications*.

Mackworth, A.; Mulder, J.; and Havens, W. 1985. Hierarchical arc consistency: exploiting structured domains in constraint satisfaction problems. *Computational Intelligence* 1:118–126.

McDermott, J. 1982. R1: A rule-based configurer of computer systems. *Artificial Intelligence* 19(1):39–88.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 25–32.

Mittal, S., and Frayman, F. 1989. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1395–1401.

Stefik, M. 1995. *Introduction to Knowledge Systems*. San Francisco, CA: Morgan Kaufmann Publishers Inc.

Tsang, E. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press.