Configuration design problem solving

B. J. WIELINGA , A. TH. SCHREIBER

University of Amsterdam, Department of Social Science Informatics, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands. E-mail: {wielinga,schreiber}@swi.psy.uva.nl

1 Introduction: the problem

Configuration design[1] is a form of design where a set of pre-defined components is given and an assembly of selected components is sought that satisfies a set of requirements and obeys a set of constraints [13,17]. Sometimes an optimality criterion may be given that defines an ordering upon possible solutions. A typical example of a configuration problem is the design of elevators from a set of parameterized components given a number of initial value specifications as requirements. This problem, initially studied by Marcus [9] and known as the VT problem, has been the subject of an extensive experiment conducted by the Knowledge Engineering community in order to compare different methods to solve the problem ("The Sisyphus-II VT Initiative", see [15] and the side bar).

As an example problem that illustrates many aspects of configuration problems in general, consider the configuration of artifacts from Lego™ blocks. Figure 1 shows five types of Lego™ blocks (A-E) from three different viewpoints. The set of components consists of any number of each of these component types. Consider the task of building a small rowing boat model from these components.

The general goal of building a boat should be translated into a set of requirements. Let us assume that the requirements specify general geometric properties of the boat as illustrated in table 1. In the example problem the constraints are on the one hand determined by the properties of the Lego™ blocks and how they can fit together, and on the other hand by constraints on the design of the boat itself: it should be symmetric, all parts of the boat should be connected and there should be no "holes" in the boat.

An optimization criterion for the example problem could be to use a minimal number of blocks for the configuration. Figure 2 shows a minimal solution to the boat configuration problem. Given the standard dimensions of Lego™ components, the requirements cannot be satisfied exactly: the solution has dimensions that are slightly different from the requirements. This is a typical phenomenon in configuration design.

[1] The web page http://www.cs.unh.edu/ccc/config/ contains a variety of links to interesting work on configuration design

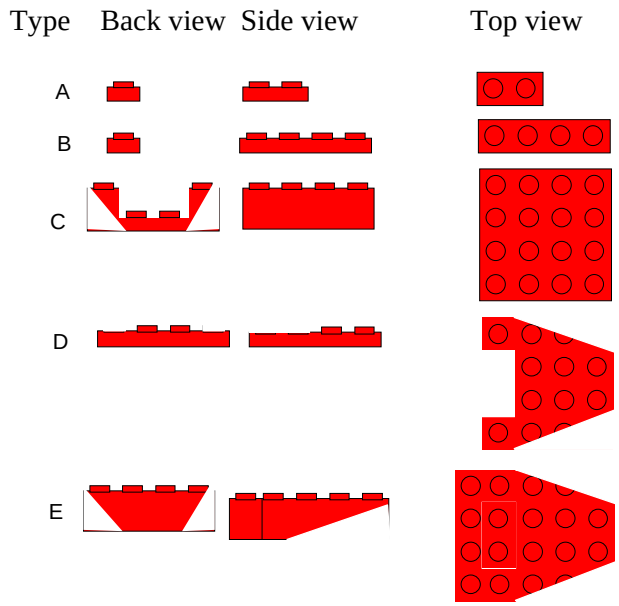


Figure 1: Components in the example problem

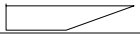
Requirement	Type	Geometry
R1	side view	
R2	hull cross section	
R3	aft view	
R4	dimensions	Width = 3 cm; Length=6 cm; Height=1 cm

Table 1 :Requirements for a Lego™ boat

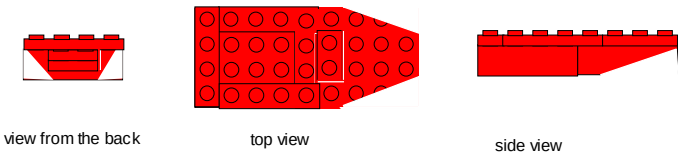


Figure 2 : A solution to the Lego™ boat problem

Many systems that solve problems that satisfy the general definition of configuration design have been described, but when one takes a closer look at the various variants of the configuration task, subtle differences exist that have an impact on what knowledge is involved and how the task can be performed.

A first source of variation is the set of components. On the one hand, the set of components can entirely be determined in advance, as is the case in the configuration of a mold for the production of a completely defined product. The components, their shape and dimensions are known and the configuration problem consists in finding the optimal layout of the components on the mold. The example domain of Lego™ boat configuration as described above, also fully specifies the components. On the other hand a significant amount of freedom may exist in specifying the components: properties of the components may have to be assigned values, components may have different instantiations, components may have a variable number of places to which other components can be connected ("ports"). In the Lego™ world, the set of basic blocks could be specified as a generalized block that is parameterized by length, width height, and color. In that case the specification of the component space would contain one type (the rectangular block) with three parameters which would replace the types A and B in Figure 1.

The relations that make up the arrangement of components provide a second source of variation in configuration design. One extreme in the spectrum of configuration problems is the case where the arrangement is fixed and the configuration task is reduced to assigning values to parameters of the components (parametric design). A less restricted case is where a skeleton arrangement is given, but the specific arrangement still has to be determined [2,7]. The other extreme case is where the space of possible arrangements is not limited in any way by the specification of the problem. The Lego™ boat problem is an example of a configuration problem where the skeleton arrangement is given, but the constituent blocks and their relations still have to be determined.

A third source of variation in the task is the degree in which the requirements can be directly related to the components and their relations. For example in the VT problem [9,15], the requirements are specified in terms of a number of initial value assignments to component parameters. Other variants of configuration assume the requirements to be functional, in which case some relation between the arrangement of components and the required function must be determined [13]. In the Lego™ boat problem, the requirements are not directly related to the components themselves, but to geometric properties of the final assembly. Testing the requirements against a possible solution will involve some aggregation over the properties of the individual components.

Constraints differ from requirements in that requirements must be satisfied, while constraints must not be violated. For example, a requirement could be that a house should have an outlet for smoke from the fire. Such a requirement could be satisfied by a chimney or some other contraption that fulfills this function. An example of a constraint would be that a house should not have a chimney higher than 1 meter. This constraint does not say that there should be a chimney, but that if there is to be one, it should not exceed a certain height. The difference between requirements and constraints is subtle and not always recognized. However, in logical terms the difference amounts to the distinction between logical consequence (for the requirements) and logical consistency (for the constraints).

Constraints can be local, i.e. applicable to a single parameter, a single component or to a relation between two components, or a constraint can be global. For example the constraint that the weight of a device must not exceed one kilogram is a global constraint since the weight of a device is not a property of any of the components. On the other hand, a constraint like "the distance between the platform edge and the hoistway wall should be larger than 8" is a local constraint between a pair of parameters. In the Lego™ boat example there are local constraints that determine how Lego™ blocks can be connected (i.e. they should have the same orientation and the knobs should align) and there is the global constraint of symmetry. and Constraints that concern the function of an assembly are also global constraints. In addition to local and global constraints, there is an intermediate group of constraints that can be termed *incrementally applicable*. These constraints are defined for a local part of the assembly, but can only be checked once other parts have been designed as well.

Configuration design differs from other types of design in two respects: no new components can be designed and the set of requirements and constraints is assumed to be complete, i.e. no requirement design is necessary. Even if hierarchies of partial assemblies are used in configuration, such hierarchies are assumed to be known beforehand. Thus, configuration design is a special case of what Brown and Chandrasekaran [3] call Class 3 design.

Although the terms *component*, *parameter* and *assembly* suggest that configuration design mainly applies to the configuration of physical artifacts, the components can also represent other things, e.g. activities. From that perspective scheduling is in many respects similar to layout design: the components (activities) are fully specified and the problem is to find an arrangement of the activities in time that satisfies a number of constraints. A planning task where the possible actions that constitute the plan are known, but where the time order and dependencies between actions are unknown, can also be viewed as an example of a configuration design task.

The set of solutions to a configuration problem is a subset of the space of all possible assemblies of components. This space is often subdivided into the space of *valid* configurations (that satisfy all constraints), the space of *suitable* configurations (that satisfy the constraints and requirements) and the space of *optimal* configurations: suitable configurations that satisfy optimality criteria. In addition to these distinctions, one also distinguishes between configurations that are constructable and those that cannot be constructed in the real world. For example, one could imagine a configuration of molecules that performs a motor function, but such a device would be very difficult to construct given state-of-the-art technology. In principle, constructability conditions could be formulated as constraints, but in practice they are often omitted.

Table 2 summarizes the dimensions on which configuration problems can differ and shows for each of these dimensions some of the values that a problem can have. A space of configuration problem types can be constructed by taking the cross product of the values on the three dimensions. Of course, much more subtle differences could be included in the table, but here we restrict ourselves to a limited range of values.

components	*assembly*	*requirements and constraints*
fixed set (c1)	fixed (a1)	local, fully and directly applicable (r1)
fixed set, but parameterized (c2)	skeleton given (a2)	incrementally applicable (r2)
set of types of parameterized components (c3)	free (a3)	functional and/or global (r3)

Table 2: dimensions of configuration design problems

Table 3 gives some examples of cells of such a cross product and the corresponding term that is often used for this problem type. The first three rows represent degenerate forms of configuration design: the entire assembly is given, the problem is to verify the validity and suitability. Assignment can be viewed as a configuration task where a given set of elements (e.g. people) have to be fitted into a skeletal arrangement (e.g. rooms). Where the components are fixed, but the arrangement is largely open, the task is usually called "layout design", or "scheduling" where it concerns the arrangement of activities in time. Parametric design assumes a given arrangement and is mainly concerned with the assignment of values to parameters of components. In skeletal design the assumption is that some abstract model of the assembly is known, but that the details of the mapping between components and the assembly are to be found.

The nature of the requirements and constraints (see Table 2, column 3) also influences the way in which configuration design can be approached. When requirements and constraints are local, i.e. apply to single components or relations, they can be verified whenever the relevant part of the configuration is known. If requirements and constraints involve a larger number of components they can be ordered with respect to the partial assemblies that they apply to and can be verified incrementally during the configuration process. Finally, if requirements apply to functional or behavioral properties of the system as a whole, they can only be tested when the full assembly has been determined.

Cell	Task category
c1, a1, r1	local verification
c1, a1, r2	incremental verification
c1, a1, r3	functional verification
c1, a2, r2	assignment
c1, a3	layout, scheduling
c2, a1	parametric design
c3, a2	skeletal design
c3, a3, r3	full configuration design

Table 3: example types of configuration design

In conclusion, configuration design spans a large space of problem types, where different types of knowledge are required and different problem solving methods are applicable. The goal of this

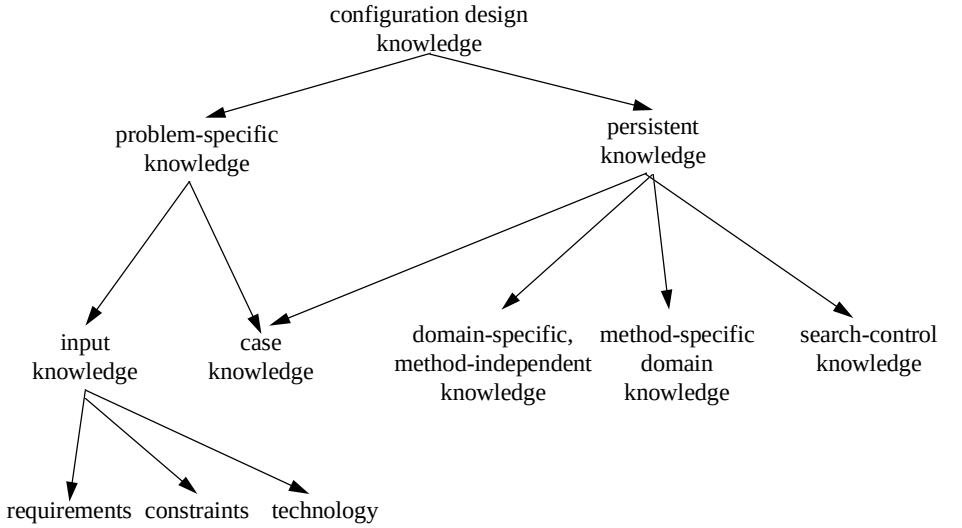


Figure 3: A general classification of knowledge involved in configuration design

paper is to investigate the different types of knowledge that play a role in configuration design and to identify some of the problem solving methods that apply to the configuration task. The result will not be a full theory of configuration design, but we hope to make some steps in that direction.

2 Knowledge categories in configuration-design problems

Before we can look at the different kinds of knowledge involved in configuration-design problem solving in detail, we first must define the major categories of knowledge involved in this task and indicate under what circumstances they are relevant. A first distinction to make is between problem-dependent knowledge and knowledge that remains valid over multiple problem solving sessions. The former category is input knowledge and the latter is persistent knowledge. A special category of knowledge concerns previously solved cases. If previous solutions are stored in the knowledge base they become persistent and can be used as a general resource in future problem solving processes. A second general distinction is between knowledge that pertains to aspects of the design domain --for example the domain of elevator -- and knowledge that is also domain-dependent, but specific for a particular problem solving method. Another category of persistent knowledge could be independent of the domain, but would be specific for a particular method of solving the problem. The latter type will be called “search-control knowledge”. Figure 3 shows a hierarchy of global knowledge categories in configuration design.

In the literature on configuration design there exists a wide variety of definitions of what constitutes the input to a configuration problem solver and what belongs to the knowledge base that persists over problem solving processes. There is general consensus that some requirements and constraints are user-supplied input. Often the assumption is made that the set of requirements and constraints as given in the input is complete. However, in realistic design tasks many requirements

and constraints must be derived from knowledge about the design domain. Moreover, certain constraints can depend on design decisions made during the problem solving process. For example, if an architect decides to place heating equipment in a certain place in a house, there is no doubt that constraints from fire-prevention regulations must apply to that place. In such cases the constraints will have to be generated dynamically during the design process [12]. Some authors include optimality criteria as part of the user-provided input. Others view information about the set of components and the possible compositional structures as part of the input specification.

Following Chandrasekaran [4] one could argue that a specification of the technology to be applied in the design process is also part of the input. In addition to the problem specification-- knowledge is needed about the design domain (*domain theory*), additional requirements and/or constraints that are not explicitly given as input but that are implicitly given in regulations, common sense etc., optimality criteria and optionally certain preferences. Preferences are design choices which are not the outcome of the design process itself, but which are given beforehand and constrain the space of design options. Such design choices may be changed when no solution can be found within the constrained search space. For example in the VT domain many parameters get values assigned within a predefined range. When these values cause constraints to be violated the range of possible values can be relaxed using “fixes”.

An essential type of knowledge for configuration design, is knowledge about components. In its simplest form this knowledge takes the form of names, representing component types, but in most cases the names are associated with properties that have pre-defined values (fixed components) or that can be assigned certain types of values (parameterized components). Constraints on the values that a property can have are usually represented as value-sets or ranges. Knowledge about components can be organized in subclass hierarchies. Relations between components (usually their connections) can take entire components as argument, or can relate ports of components. In the latter case, knowledge about ports -their type and number- is part of the component knowledge. The components from which an assembly is build can be represented as instances of component types. This causes the issue of cardinality to become important: how many instances of a component should be part of the assembly? The cardinality of components can be derived in various ways: it can be part of the requirements (“this car should have four wheels”), it can be specified as a part of given knowledge about the assembly (“ a car consists of four wheels and ...”), or it can be the result of a problem solving process.

In many domains, in particular in technical ones, knowledge about partial assemblies of components is available. Such assemblies could be viewed as aggregate components, provided that their properties and ports are known and that they satisfy all local constraints that apply to their parts. In the context of configuration design, we assume that partial configurations are fixed or parameterized.

Knowledge about the connection relations that can exist between components can take various forms. First, knowledge is required about the relation itself: name, arity, argument types, constraints on arguments etc. This information constitutes the *signature* of the relation. Second, a relation can have properties. For example, if we have a relation that represents the design choice that two components are next to each other, we may want to add the distance between them as a property to the relation. Third, relations can have a hierarchical sub/super-type structure. Subtype relations are more commonly used for concepts than for relations, but a type hierarchy of relations can play an important role in configuration design. For example, a connection between two electrical components may be needed to satisfy a requirement. Representing the connection in

general terms may be enough to reason about the requirements, but more details about the nature of the connection may be needed for the solution of the configuration problem. A connection between two electrical components can be made by a wire, a bus, a plug, infra-red channel, laser signals, etc. Subordinate relations of the general *connect* relation can be used to further specify the nature of the connection.

In many configuration domains, some knowledge about the global structure of the assembly is known in advance. In designing a car, the designer will have a skeletal assembly in mind consisting of component types such as wheels, engine, steering system etc. Such skeletal assemblies are often formulated in terms of general component classes, rather than in specific instances of components. For example, in the VT domain the general structure of an elevator system is given, but choices have to be made how component classes will be specialized and instantiated. Yu and MacCallum [17] introduce the similar concept of "Product Breakdown Structure" which in essence is a part-of tree of component classes.

Mittal and Frayman [13] define a functional architecture as a decomposition of an artifact in terms of functions, rather than in terms of components as is the case in skeletal assemblies. However, the boundary between functions and components is fuzzy. Runkel *et al.* (in [15]) use the term "function" to refer to general types of components such as *motor* and *door* in the VT domain, and reserve the term component for those parts of the artifact which have no further decompositions. The Ontolingua ontology for the VT domain uses the term component for both cases (see [15], Gruber *et al;*). This terminological confusion arises because components -viewed as physical objects- are often classified according to their function [1]. Although a purist would require a clear separation between function and component, the merge of the two notions in a functional abstraction hierarchy of components can have its advantages for problem solving as Mittal and Frayman show.

The term "constraint" is used in a variety of ways. The VT ontology uses the notion of constraint for all forms of mathematical or logical relations that holds between parameters. Others distinguish between expressions the can be used to compute the value of a parameter and proper constraints that just test whether parameter values are within certain boundaries. Secondly, there is a distinction between constraints that only involve values of parameters (*local constraints*) and those that involve parameters that are not associated with one component, but with a larger part of the assembly. Finally, constraints are sometimes used to represent preferred strategies rather than hard constraints that are inherent to the domain. Motta *et al.* (in [15]) argue in favor of a clear separation between the inherent constraints and other types of expressions that have similar syntactical structure, but that have a different underlying rationale. Yu and MacCallum [17] distinguish between constraints that represent certain dependencies between design choices --one design choice forces another-- and constraints that act as limitations in the space of design choices.

Many systems use degenerate representations of the knowledge described above. For example, in the VT problem component selection is represented as value assignment and a property "model" is used to represent subtypes of components. Although such representational choices can make the problem solving process for configuration design easier or more uniform, they also cause a lack in ontological clarity, which can have negative effects on knowledge maintenance and reuse.

Figure 4 shows the consists-of hierarchy of domain-specific, method-independent knowledge categories. In addition to the domain-specific knowledge a configuration-design system may need

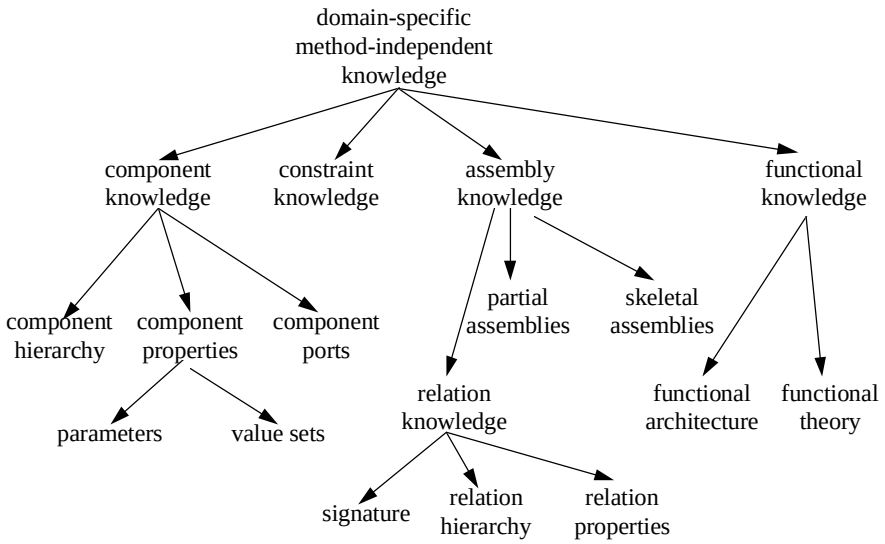


Figure 4: Consists-of hierarchy of domain knowledge in configuration design

additional knowledge to reduce the search space and to control the problem-solving process. These categories of knowledge will be discussed in the following section.

3 Methods for solving configuration-design problems

Given a knowledge base that contains the knowledge types described in the previous section, the configuration-design task could -in principle- be solved by a straight forward search process. However, the combinatorial nature of the problem makes this option an unrealistic one in practice. Hence, problem solving methods (PSMs) are needed that in some way or other constrain the search process by using knowledge -heuristic or otherwise- and by introducing additional assumptions that restrict the search space. Several authors have given overviews of PSMs for design and more specifically configuration. Puppe [14] describes three main PSMs for construction problems: heuristic construction, model-based construction and case-based construction. Heuristic methods for configuration include: skeletal construction, propose-and-revise and least-commitment (see below). Chandrasekaran describes a number of PSMs in the context of a general family of methods based on a top-level decomposition of the design task into three sub-tasks: Propose, Critique and Modify (PCM). Maher [10] describes three process models of design: decompositional design, case-based reasoning and transformational design. Löckenhoff & Messer [8] present detailed KADS models for case-based configuration, a structure-oriented approach where a taxonomical structure of components is mapped onto a graph, and a resource-oriented approach based on balancing of resource requesting and production model of components. Even though there is no

clear consensus on a taxonomy of PSMs for the configuration-design task, there is clearly some overlap.

All of the PSMs discussed above are knowledge intensive: they make heavy use of knowledge to constrain the configuration process. Other classes of methods are based on uniform reasoning procedures combined with specialized representations of the problem to be solved. The vast literature on solving constraint problems falls in this category. Many configuration-design problems can be formulated as a constraint satisfaction problem (CSP). Also, most working systems use ideas that were first developed in the CSP literature. However, pure CSP methods often fail to make effective use of knowledge about the domain at hand. Also, the constraint-satisfaction methods are unable to explain the rationale behind the solution found. There is a trade-off between representational transformations and uniform methods on the one hand, and more complex representations and consequently more complex reasoning procedures on the other hand. The combinatorics of complex problems can best be handled through the use of domain-specific knowledge.

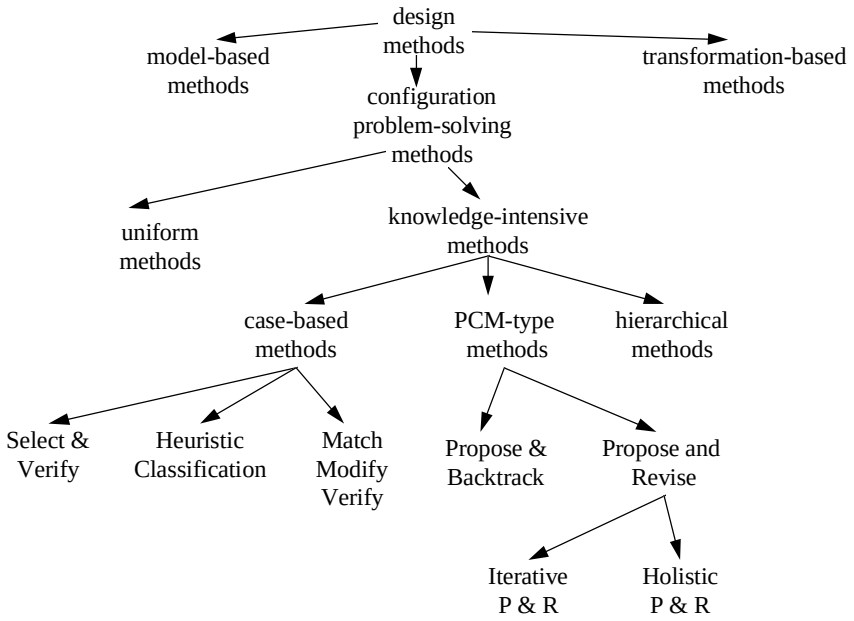


Figure 5: A partial taxonomy of PSMs for design and configuration

Figure 5 shows an attempt to classify some PSMs for configuration design. At the top level we distinguish between methods that make extensive use of knowledge and those based on domain-independent uniform methods, such as constraint satisfaction and linear programming. Transformational and model-based methods are can be applicable to configuration design, but have a wider scope. These methods are covered by other papers in this issue (i.e. Grammatical Design, Functional Design). Here. we focus on those methods that are knowledge intensive.

3.1 Case-Based methods

This class of methods is based on the assumption that knowledge about solutions of configuration problems is explicitly represented. Such knowledge can be based on previous solutions, or can just be obtained through knowledge acquisition from human experts.

A simple variant of configuration design that can be handled with case-based methods, arises when a set of arrangements is given from which a potential solution can be selected, which subsequently needs to be verified against the problem-dependent requirements and constraints. This simple form of generate-and-test, which we call *select-and-verify* requires knowledge that can guide the selection process -how to find the configuration that corresponds best with the current problem- and knowledge about the verification process. The selection process can be simple or can be based on a more complex matching process where requirements are matched against properties of the given arrangements. Clancey [5] proposes heuristic classification as a way of selecting and verifying enumerated solutions to configuration problems.

The select-and-verify PSM assumes that a solution can be found among the given set of arrangements. In general case-based reasoning this assumption is relaxed such that a solution can be found by modifying a given configuration. Two aspects of case-based configuration are problematic: finding the best candidate from the set of given configurations and blame assignment, i.e. identification of those aspects of the stored solution that cause violations of constraints and requirements. See the paper on case-based design in this issue for more details on the case-based design method in general.

3.2 Propose, Critique and Modify (PCM) methods

The class of methods based on the PCM approach, was extensively discussed by Chandrasekaran [4]. The general principle is that an initial configuration is proposed, which is subsequently tested against constraints and requirements. If violations are found, modifications are used to remove the conflicts. Several specializations of the general PCM method have been used for configuration design. We discuss two of these in more detail.

Propose-and-backtrack

Propose-and-Backtrack is an iterative method where a design decision is made and subsequently tested against the requirements and constraints. If the test fails alternative decisions are considered. If no viable alternatives can be found, chronological backtracking is used to undo earlier decisions. The major difference between Propose-and-Backtrack and straightforward depth-first search is the propose step. Knowledge about the domain can inform the propose step to select the most appropriate decision first. The initial version of the DIDS solution to the VT problem (see [15], Runkel *et al*,) combined this method with constraint propagation. A major problem with this method is its inefficiency. In the VT domain this method appeared to be about one order of magnitude slower than other methods.

One could argue that R1/XCON [11] uses a form of Propose-and-Backtrack. The fact that R1 performs little search (backtracking) arises because R1 possesses crucial knowledge about what to do next. A large fraction of R1's knowledge concerns the decision which sub-task in configuration-design is the most appropriate one to do next, such that search will be minimized (McDermott,

1993). This observation appears to generalize: the order in which design extensions are proposed is crucial for effectively solving configuration problems (see [15], Yost & Rothenfluh). The Least-Commitment method [6,16] is another way of reducing the backtracking search. The principle behind the Least-Commitment method is to postpone design decisions where possible and leave many options open. Pending decisions are transformed into constraints that reflect the dependencies between parameters rather than actual values. In its general form Least-Commitment is not a knowledge-intensive method and systems based on this method may have difficulty recovering from wrong decisions.

Propose-and-revise

Propose-and-revise (P&R) is the method used in the original VT system [9], and subsequently studied extensively in the Sisyphus-VT study. This method decomposes the configuration-design task into three sub-tasks: (I) *propose* a design extension, (ii) *verify* the current design, and (iii) *revise* the design, if necessary. The main difference with propose-and-backtrack is that P&R does not undo previous design decisions, but instead "fixes" them.

P&R does not require an explicit description of components and their connections. The method basically operates on one large bag of parameters. Arrangement information thus has to be represented as a (special type of) parameter.

Invocation of the propose task does not produce a full design, but the smallest possible extension of an existing design, namely one new parameter assignment. The order in which parameters are selected is guided by domain-specific search-control knowledge, which provides an ordering of parameters based on the components they belong to.

The verification task in P&R applies a simple form of constraint evaluation. The method used here is to perform domain-specific calculations provided by the constraints. The word "constraint" in P&R has the restricted meaning described in the previous section: a formula that delivers a Boolean value.

The revise task in P&R implements a quite specific strategy for modifying the current design, whenever some constraint violation occurs. To this end, the task requires a knowledge about fixes. Fixes are a second form of domain-specific search-control knowledge used by the method. Fixes represent heuristic strategies for repairing the design in case of a constrain violation. Fixes also incorporate design preferences. The goal of the revise task is to change the current design in such a way that it is consistent with the violated constraint. This goal is realized by applying combinations of fix operations which change the value of parameters, and subsequently propagating these changes through the network formed by the computational dependencies between parameters. The application of a fix may introduce new violations. Fixing these new violations would involve a recursive invocation of *revise*. P&R tries to reduce the complexity of the configuration design by disallowing recursive fixes. Instead, if the application of a fix introduces a new constraint violation, the fix is discarded and a new combination of fixes is tried.

Motta *et al.* (in [15]) have pointed out that in terms of the flow of control there are two possibilities in P&R. One can either perform verification and revision either directly after every new design or after all parameter values have been proposed. The first strategy was used in the original VT

system, but Motta argues that the second strategy is more efficient, and also comes up with a different set of solutions.

Although this method has been proven to work in practice, it has inherent limitations. Using fix knowledge implies that the design revisions are guided by heuristic strategies. Fix knowledge implicitly incorporates preferences for certain designs. This makes it difficult to assess the quality of the final solution produced by the method. .

3.3 Hierarchical Configuration

Several variants of a PSM that employs some form of hierarchical decomposition exist. In its basic form hierarchical configuration is a form of AND/OR graph search. The top-level goal is decomposed into a number of alternative substructures, each of which represents a potential refinement of the original goal. Variants of the hierarchical decomposition method differ in the type of knowledge that is represented in the nodes of the decomposition tree and in the way the tree is constructed or traversed.

Several variants of this approach exist including:

- named nodes in the AND/OR tree
- use of a functional architecture : the AND and OR parts of the tree are separated in different knowledge structures
- use of design plans (with some additional knowledge about how to handle constraint violations)
- problem decomposition versus device decomposition

A design plan is a knowledge structure that describes how a particular requirement or sub-problem can be solved. An explicit notion of configuration through design plans is a feature of DSPL [3]. This method assumes localized knowledge for handling constraint violations. It presupposes a functional architecture with a mapping of design functions to components. This architecture is not limited to configuration design, but can also be used for other design problems.

The hierarchical methods for configuration-design should be preferred, if the arrangement of components is an important element of the application problem (i.e. if there is not a single, fixed skeletal design). This will in practice a major selection criterion between, for example, P&R and DSPL.

4 Conclusions

Although configuration design seems to be one of the simpler forms of design problem solving, it still involves many different categories of knowledge and is amenable to many different problem solving approaches. Even the simplest forms of configuration design, such as parametric design and layout design, require advanced AI technology to be solved.

The required domain-knowledge structures for the various approaches to configuration design differ significantly. Many authors prefer a parsimonious set of knowledge structures that suit their favorite problem solving method or their domain. In the future, much can be gained by making these ontological commitments explicit and clearly distinguishing between domain-dependent and method dependent knowledge categories. Knowledge reuse and integration of different methods in one system should benefit from such explications of the ontological distinctions. Parsimony and representational efficiency can still be achieved through mappings between a general ontology for configuration design and an ontology for a particular application (see [15], Schreiber & Terpstra)..

Two major approaches to problem solving in configuration design can be distinguished: the use of domain-independent, uniform methods and knowledge-intensive methods. Although the former approach has the advantage of a wide applicability, it remains based on weak methods. Clever techniques, good human-computer interfaces, well engineered algorithms and formal theories can make these methods very attractive and even efficient in certain problem areas. However, many realistic configuration-design problems remain where these methods become unacceptably inefficient due to the combinatorial nature of the domain.

The knowledge-intensive approach has proven to be successful, even though constructing and maintaining a particular application may be time-consuming and expensive. The most salient feature of the knowledge-intensive approach to configuration-design is that the order of design decisions is a crucial one. This is the lesson drawn from the long-standing experience with R1/XCON [11] as well as a general conclusion from the Sisyphus-VT effort. The current knowledge about ordering these decisions is entirely domain specific. An issue for further research would be to see whether such knowledge can be reformulated or generalized to become applicable to a wider class of configuration-design problems, and even can become reusable.

Propose-and-Revise is a promising method where the main problem is to find value-assignments to parameters. If component selection is of major importance, hierarchical methods appear to be good candidates. An issue that remains to be decided is whether repairing a wrong configuration proposal is easier or more difficult than designing a good one in the first place. There seem to be some indications -both from competent AI systems such as R1 and from human experts- that getting it right in the first attempt, using a significant amount of domain specific knowledge, is the most optimal route towards efficient and competent configuration-design systems.

Acknowledgments

Discussions with J.M. Akkermans are gratefully acknowledged. We like to thank Dave Brown, Bill Birmingham and three anonymous reviewers for their helpful comments on earlier versions of this paper.

The work reported here was partly carried out in the course of the KACTUS project. This project is partially funded by the ESPRIT Programme of the Commission of the European Communities as project number 8145. The partners in the KACTUS project are ISL (UK), LABEIN (Spain), Lloyd's Register (United Kingdom), STATOIL (Norway), Cap Programmator (Sweden), University of Amsterdam (The Netherlands), University of Karlsruhe (Germany), IBERDROLA (Spain), DELOS (Italy), FINCANTIERI (Italy) and SINTEF (Norway). This paper reflects the opinions of the authors and not necessarily those of the consortium.

References

[1] Benjamin, J., Borst, P., Akkermans, J. M., and Wielinga, B. J. (1996). Ontology construction for technical domains. In N. R. Shadbolt, K. O'Hara and A. Th. Schreiber (eds.) *Advances in Knowledge Acquisition: Proceedings Ninth European Knowledge Acquisition Workshop EKAW'96*, pp. 98–114, Lecture Notes in Artificial Intelligence, volume 1076, Berlin/Heidelberg, Germany, May 1996. Springer Verlag.

[2] Birmingham, W. P., Gupta, A., and Siewiorek, D. (1992). *Automating the Design of Computer Systems: The Micon Project*. Boston, Jones, and Barlett.

[3] Brown, D. C. & Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*. Research Notes in Artificial Intelligence, London, Pitman.

[4] Chandrasekaran, B. (1990). Design problem solving: A task analysis. *AI Magazine*, Winter issue, pages 59–71.

[5] Clancey, W. J. (1985). Heuristic classification. *Artificial Intelligence*, **27**:289-350.

[6] Descotte, Y. and Latombe, J. (1985). Making Compromises among Antagonistic Constraints in a Planner. *Artificial Intelligence* 27, 183-217.

[7] Friedland, P. and Iwasaki, Y. (1985). The Concepts and Implementation of Skeletal Plans. J. of Automated Reasoning **1**:161-208.

[8] Löckenhoff, C. & Messer, T. (1994). Configuration. In: Breuker, J. & Van de Velde, W., *CommonKADS Library for Expertise Modelling*. Amsterdam: IOS Press, pp 197-212.

[9] Marcus, S., Stout, J. & McDermott, J. (1988). VT: An Expert Elevator Designer that Uses Knowledge-Based Backtracking. *AI Magazine*, Spring issue, pages 95–111.

[10] Maher, M. L. (1990). Process Models for Design Synthesis. *AI Magazine*, Winter issue, 1990, pp 49-58.

[11] McDermott, J. (1993). R1 ("XCON") at age 12: lessons from an elementary school achiever . *Artificial Intelligence* 59:241-248.

[12] Mittal, S. & Falkenhainer, B. (1990). Dynamic Constraint Satisfaction Problems, *Proc. 8th Nat. Conf. on AI, AAAI-90*, 1990, pp. 25-32.

[13] Mittal, S. & Frayman, F. (1989). Towards a generic model of configuration tasks. In *Proceedings of the 11th IJCAI*, pages 1395–1401, San Mateo, CA, Morgan Kaufman.

[14] Puppe, F. (1993). *Systematic Introduction to Expert Systems*. Berlin: Springer-Verlag.

[15] Schreiber, A. Th. & Birmingham, W. P. (1996). The Sisyphus-VT initiative. Special issue of *Int. J. Human-Computer Studies*, 44(3/4)

[16] Stefik, M. (1981). Planning with Constraints (MOLGEN: PART-I). *Artificial Intelligence* 16, 111-139.

[17] Yu, B. and MacCallum, K. (1995). Modelling of Product Configuration Design and Management by Using Product Structure Knowledge. *Int. Workshop on Knowledge Intensive* CAD, Finland, 1995.

Side bar: to be placed somewhere around the second page

The Sisyphus-VT study

In 1992, the knowledge acquisition community decided at their annual workshop in Banff (Canada) to conduct a comparative study of existing knowledge engineering methods and techniques. The method used was to request people to build a system using a common data set. For this purpose, the elevator-design application was chosen. This domain had been the target of the VT application developed by Marcus, Stout and McDermott [9]. A prime reason for this choice was the availability of a meticulously documented description of the original application data, written by Gregg Yost for the purpose of his Ph.D. work at Carnegie Mellon. Also, Tom Gruber and Jay Runkel added a predefined ontology for configuration design, plus a corresponding knowledge base for the elevator domain.

The results of this study, which is part of a series of studies named "Sisyphus", can be found in a special issue of the *Int. J. of Human-Computer Studies* [15]. Although actually more as a side effect, the papers in this issue contain a wealth of information about how to model practical configuration design problems,. The issue also contains the original data set and the proposed configuration-design ontology.