From: AAAI Technical Report FS-96-03. Compilation copyright © 1996, AAAI (www.aaai.org). All rights reserved.

Dimensions of Configuration Problems – Looking for a Unified Configuration Model

Markus Stumptner
Christian Doppler Laboratory for Expert Systems
Institut für Informationssysteme, Technische Universität Wien
Paniglgasse 16, A-1040 Vienna, Austria
Tel: (+43 1) 58801-6129 Fax: (+43 1) 505 53 04
e-mail: mst@dbai.tuwien.ac.at

Abstract

Configuration design deals with the composition of artifacts from a predefined catalog of components over a wide variety of application domains (architecture, telecommunications systems, civil engineering, control software, etc.). Recent years have seen various attempts to work towards a high-level, formal representation of configuration problems with the goal of providing a stable platform on which generally usable configuration tools can be built and meaningful theoretical research results achieved. Unlike other application areas that are amenable to a model-based approach (such as diagnosis), so far no scheme has been generally accepted to provide such a unified basis. In this paper, we analyze different modelling options in CSP-based configuration systems, discuss their realization in particular in the realm of dynamic CSP's and our own generic CSP approach, and try to draw a bridge to several predominantly logic-based approaches to configuration by discussing them in view of our criteria.

Introduction

Configuration systems have traditionally been among the most successful AI applications all the way back to the venerable R1/XCON system (McDermott 1982) whose descendants are still in use today. This is probably due to the fact that configuration domains are usually well-structured, providing an explicit description of the entities of the problem domain (such as a parts catalog for a technical system), the properties and behavior of these entities, i.e., the ways in which they interact, and of the goals and parameters of individual configuration problems (cost, performance, weight, etc.).

As the maintenance and modeling problems engendered by the use of rule-based systems became visible (Barker and O'Connor 1989), researchers started to look for a better suited, encompassing representation that could be used to represent all kinds of configuration problems. This search is essentially still going on. However, constraint satisfaction problems (CSP) (Frayman and Mittal 1987), have found widespread acceptance (see, e.g., (Steinberg *et al.* 1989; Faltings and Weigel 1994)). We will try to give a brief classification of some dimensions that influence the representation of configuration domains based on the CSP paradigm.

A CSP View of Configuration

The basic assumption we make is that the system to be configured will be composed of a number of well-defined components that interact with each other. Components will be selected to take part in the configuration depending on the function they provide for the finished system. It should be noted that this is a general enough formulation to cover such diverse areas as telecommunications equipment, process control software modules, and restrooms in office or tenement buildings, to name a few examples. Certain combinations of components may be illegal due to technical, geometrical, legal or other requirements (e.g., using AC vs. DC power supply, putting a bathroom in a space where it will be too short for the bathtub, using a warning light that does not fit a certain nation's safety standards). It is a convenient choice to formulate such restrictions as constraints.

A CSP is defined by a set of variables V, with a domain $dom(v)$ associated with each variable. Constraints are relations defined on groups of variables that specify what combinations of values may be assigned. A variable x can be considered as a *place* in a system where one the components from $dom(x)$ can be used, or as a *port* where one component can be plugged into another. The constraint-based approach is generally accepted to offer certain advantages (Stumptner *et al.* 1994):

- Constraints allow clear separation of problem and strategic knowledge, since strategies can be expressed declaratively, in terms of variable and value orderings (e.g., fill a frame starting with the leftmost

slot) without changes to the constraints themselves.

- A constraint-based representation can be used for either synthesis or verification: Given a finished configuration in terms of the constraint network that represents it, it is possible to check all constraints for inconsistency and thus test the validity of the configuration.

- The simplicity and declarative nature of the basic constraint model allows the definition of clear semantics.

- A large body of work exists on efficient reasoning with CSPs.

A Catalog of Representation Choices

Starting from the basic concepts described above – choosing a type of component and choosing a component to connect to another, we now consider a number of variants of the problem and how they influence the desired expressiveness of a constraint language. We consider them in the order in which they lead us farther away from the original CSP view.

1. **Atomic vs structured components.** As "atomic" we consider here a component which has no properties that are not determined by its type. In this case, assigning just the type to a port where a certain component is needed is fully sufficient (e.g., *battery=large*), and individual components need not be distinguished. If similar components in different parts of the system can have individually assigned property values, a special domain is needed to determine which individual component is assigned to a port, (e.g., $battery = b051$, $size(b051, large)$, $ship\_date(b051, 960602)$). In effect, we are introducing explicit object identity for components.

2. **Individual properties referring to components instead of just values.** The example above describes batteries with date-valued individual property variables. However, since we have allowed component identifiers as domain values above, we can now just as well allow properties that have components as domain. In this environment the term "port" takes its full meaning. A port now is no longer just a fixed position in the system where some other component is added (e.g., "the printer cable"), a port now is a connection between two components that each have a separate existence. In addition to object identity, we have now introduced what amounts to object structure. However, there exist cases where this is not enough.

3. **Multiple bidirectional connections between a pair of components.** Imagine a video and two stereo sound connectors between a TV set and the cable leading to a VCR. In this case, it is important to know which end of the cable goes into which slot on the TV. Either each line of the cable is represented as an individual component, or a special predicate is needed to describe such a connection.

4. **Aggregate Functions.** Given that we have included sets of components as variable domains, the idea suggests itself introduce constraints on these sets as well, e.g., by computing aggregates: "No more than a fifth of the space of each floor should be taken up by corridors", or "the total power output in the system must be higher than the total power consumption". The size of the set must then be balanced against its constraints – the set has become a *resource* and the constraints describe the consumption relationships for the resource.[1] This is similar to the notion of cardinality constraints introduced in (Van Hentenryck and Deville 1991).

The last criterion leads us to the notion that the set of components might itself be variable (since it is a set of domain values, it can be considered a resource). We will consider this in more detail.

Dynamic Constraint Satisfaction

The notion of a dynamic CSP was first used in a configuration context by Mittal and Falkenhainer (Mittal and Falkenhainer 1990), aiming to overcome the static nature of standard CSPs due to their fixed set of variables and constraints. Configuration problems often do not involve a fixed number of parts. In (Mittal and Falkenhainer 1990), this notion was expressed as the 'relevance' of parts; for example, if a car to be configured does not include a sunroof, then it is irrelevant whether the sunroof can be ordered in clear or tinted glass or what kind of electrical or mechanical operation could be chosen. Meta constraints are used to explicitly activate variables that need to be part of the solution. This construction saves space and time during the search for a solution and also conveys information on the structure of the constraint net.

However, this notion alone does not yet provide the advanced features described in the previous section, in particular a truly variable number of components which are often required in applications, e.g., the number of control components in a telecommunications system (Stumptner *et al.* 1994), where a configuration can include several thousand basic modules, of subtly

[1] Criteria 3 and 4 do not rely on each other, they are orthogonal additions to the previous stages.

different versions that influence how they can be connected and arranged inside the functional and physical hierarchies. The number of components can usually not be accurately predicted, as the problem parameters are often not expressed in part counts but instead in terms of global functionality. The additional step needed to produce required means of expression lies in the introduction of component identities as a separate domain.

Generative Constraint Satisfaction

These considerations are reflected in the definition of Generative Constraint Satisfaction Problems (GCSP), a name chosen because they allow variables, representing components, to be generated as needed. Also, the constraints, which are mostly associated with component types, are generic and apply separately to all the existing components of a given type (see (Stumptner and Haselböck 1993) for details). In other words, for a newly generated component of a given type, specific instances of generic constraints are created as well.

A GCSP has as a special domain the set of component variables $\mathcal{C}$. The domain of each variable from $\mathcal{C}$ is a finite set $T = \{\tau_1, \ldots, \tau_t\}$ of component types. There also exists a set $D = \{\mathcal{C}, D_1, \ldots, D_n\}$ of atomic (i.e., numeric or symbolic) domains and a function $Dom : P \mapsto D$ (P is the set of property names). The sets $D_1, \ldots, D_n$ are called *atomic domains* (i.e., they contain numeric or symbolic values, but are disjunct from $\mathcal{C}$). Note that the definition of $\mathcal{C}$ corresponds to case 1 of Section .

Both components and their properties are represented by constraint variables. The type of a component is specified by assigning a type symbol from T to that component variable. The properties associated with a component are determined by activation constraints. The name of a property variable is composed from the name of the originating component and the property name, e.g., if property $p \in P$ is active for component $c \in \mathcal{C}$, then the property variable is named $c.p$. *Require Variable* meta constraints as in (Mittal and Falkenhainer 1990) are used to describe this relationship ("if component C has type τ assigned, a component variable C.p is active"): $C{=}\tau \overset{\text{RV}}{\Longrightarrow} C.p$

The set of all possible property variables can thus be intensionally specified as $\mathcal{P} = \{c.p \mid c \in \mathcal{C}, p \in P\}$, and the domain $dom(c.p)$ of each property variable $c.p$ is defined by $dom(c.p) = Dom(p)$. According to the definition of Dom, $dom(c.p)$ can be either an atomic domain D_i, in which case we call $c.p$ an *attribute* of c, or $dom(c.p) = \mathcal{C}$. In the latter case, we call $c.p$ a *port* of c. The assignment of a component c' from $\mathcal{C}$ to a port variable $c.p$ represents a connection between c and c' via the port p of c. Note that this definition corresponds to case 2 of Section .

The following would be a typical case of a compatibility constraint:

Example: A module must not be simultaneously be mounted on $slot_1$ and $slot_2$ of a frame:

$F{=}frame \wedge F.slot_1{=}M_1 \wedge F.slot_2{=}M_2 \;\rightarrow\; M_1 \not\equiv M_2$.

As in (Mittal and Falkenhainer 1990), we assume a set of initial variables and constraints that grows as search for a solution progresses.

Finally, we discuss a facility for explicit reasoning on the existence of individual components corresponding to case 4 in section . In principle, the assumption is that the constraints defined on a particular class of domain objects will reflect the salient points of the behavior of these objects. Now, the salient points of the behavior of components are twofold – first, their type assignments and subsequent creation of property variables, and second their very existence, the fact that they are created at all. Therefore, it is reasonable to provide a mechanism for dealing explicitly with this domain.

Many configuration tasks require reasoning on *sets* of components. In standard constraint satisfaction theory this can be done by stating constraints on sets of variables, i.e., defining an n-ary constraint if there are n variables involved. Because the variables are fixed, no problem arises. In dynamic CSPs with an unlimited number of variables, where the selection of an appropriate set of variables for the solution is a problem-specific task of inference, expressing constraints on sets of variables needs a separate mechanism. *Resource constraints* provide a restricted language for expressing aggregate properties of sets of components (such as the total area of all rooms on a floor of a building). They are defined by applying a (monotonic) aggregate function to property values of a subset selected from the set of components. Proper attributes to be used for computing the aggregate function need to have non-negative domains.

Example: Corridors should not sum up to more than a fifth of the space of each floor.

$F.Area * 0.2 \geq \Sigma\{Room.area \mid Room = corridor \wedge Room.floor = F\}$

A resource function can, in principle, be used in constraint literals in every place where a constant numeric value can occur. A constraint that contains a literal with a resource function is called a *resource constraint*. Consistency is defined for resource constraints as for ordinary constraints. The difference lies in the fact that satisfying a resource constraint may place a requirement on the size of the set of components.

Thus, a solution to such a problem will by definition contain the required number of components.

The GCSP concepts described in this section were originally developed in the context of the COCOS research project (Stumptner *et al.* 1994) and are incorporated in the configurator implemented during the project. The COCOS system, implemented in Objectworks\Smalltalk, uses the GCSP model as the basis for an object-oriented constraint language (LCON). The system therefore combines an object-oriented view and implementation of constraints (as in (Caseau and Puget 1994)) with an object-oriented model of the domain. LCON uses a type hierarchy to make limited use of abstraction techniques. It should be noted that in technical domains, a taxonomy is usually provided from the start due to the organization of the original parts catalog, but experiential (heuristic) knowledge usually will be specific to individual types.

Functionally, the system is divided into a parser that translates the language into the internal dynamic constraint network, the constraint solver (conventional backtracking-based, with limited extensions for the dynamic aspects), a graphical interface for starting, controlling and viewing the constraint satisfaction process, and various utilities. The user can interactively query and modify the state of the constraint network during the solution process.

The system is in principle domain-independent, but is oriented towards symbolic domains due to envisioned application areas and does not provide for the solving of arithmetic constraints except for aggregate functions. Knowledge bases for telecommunications systems and audiovisual equipment were developed. During testing, the system solved configuration problems with about 3000 components (about twice as many variables and constraints) in the space of 10-15 minutes. This degree of performance was made possible by the fact that the real-world examples that served as basis for the experiments were relatively underconstrained, a property they seem to share with many technical configuration problems (Wright *et al.* 1993). The fact that a high number of solutions exists for a particular problem (in particular since one is probably only interested in the "best" solution – listing all 100,000 or so valid configurations for a individual set of parameters makes no sense for the configuring engineer or marketing employee) and thus makes the common domain-independent CSP algorithms quite effective.

The work of Faltings et.al. (Hua *et al.* 1990; Faltings and Weigel 1994) shows that DCSP are also useful for problems over continuous domains or mixed discrete and continuous domains, an area which was not covered in the COCOS project. This is particularly important in domains dealing with spatial requirements. Just as in conventional CSP's, the use of continuous domains requires special solution algorithms. DCSP problems are represented as subsets of a multidimensional variable space, where the dynamic activation and deactivation of constraints can lead to different subsets being active over time during the search process. Implicitly, the features of the GCSP approach are also present in these papers, as for example the number of spans of a bridge is flexible and controlled by meta constraints (Hua *et al.* 1990). The current use of a JTMS-based implementation representing constraint activations via the TMS labels of the constraints underlines our thesis about the flexibility of the DCSP/GCSP approach regarding the use of different inference mechanisms.

Logic-based configuration systems

Apart from inference mechanisms implemented in Prolog, a large part of logic-oriented configuration research uses description logic formalisms (i.e., KL-ONE-like languages). For example, the PROSE system (Wright *et al.* 1993) which is used for configurator development at AT&T, is based on the C-CLASSIC terminological language, and is commonly known as the first industrial application of such a language.

¿From the view of our criteria from Section , the emphasis on terminological logics is interesting, since these languages indeed encompass much of the expressive means listed there – an object-oriented representation with inheritance and powerful mechanisms for describing object relationships.

In addition, they share with the CSP approach the separation of domain and control knowledge and the existence of a large body of theoretical research as a solid foundation for application semantics. Much of the research in this area, however, focuses on T-Box issues (class-level) and instance-level reasoning has only recently become a more active topic.

The *Constructive Problem Solving* (CPS) approach described in (Klein *et al.* 1994) attempts to provide another encompassing formal basis for configuration design. This approach uses a feature logic as knowledge representation language and expresses the configuration problem as the problem of model construction in this language. The authors themselves note that "a quite complicated theoretical 'apparatus' [...] has been developed". This system also possesses the expressive properties from Section with the exception of case 4. It distinguishes between domain (here called "object-level") and control knowledge, and between general domain knowledge and case-specific knowledge. The authors note that their approach has some simi-

larity with the area of feature logics/terminological logics and results from this field can thus be incorporated into their system via extensions of their first-order representation language.

Resource-based systems

We conclude by looking at some examples of purely resource-based approaches (Heinrich and Jüngst 1991; Simonis and Cornelissens 1995) where components are considered as sources of a certain functionality that can be quantified, and configuration consists mostly of balancing the resource requirements against the available sources. The actual topology, i.e., identification of connecting ports between components is abstracted as far as possible, fitting case 4 but only in a limited manner to case 1 or 2. The approach from (Heinrich and Jüngst 1991) is reported to be in practical use and possesses other interesting features (e.g., a separate inheritance hierarchy for functionalities in parallel to the one for components). A simple balancing algorithm as basic reasoning method aids performance. Where necessary, reasoning about connections between individual components (where do I plug x into y) is done using a combination of production rules and constraints.

Conclusion

In this paper, we have attempted a brief overview of contrasting approaches to configuration, based on criteria which indicate the distance of configuration problems from a simple CSP representation. On one hand, the constraint-based approaches provide what could be considered a minimalist approach for representing configuration problems. The CSP approach provides independence of representational and inferential knowledge while maintaining conceptual simplicity and offering the existence of a large amount of work on efficient implementations. Systems are described and built by limited extensions to the constraint paradigm with dynamic constraints and their close relatives, generative constraints. Due to the smooth integration of the "bottom-up" alterations into the standard CSP paradigm, extensions for individual domains (e.g., continuous variables) can be accommodated. On the other side, logic-based approaches seem to take a "top-down" route where powerful languages are used from the outset and then weakened to avoid unnecessary computational overhead and complexity in description. At least when considering our criteria, it appears that in the area of applications, where these approaches meet, the gap has become quite narrow.

References

Virginia Barker and Dennis O'Connor. Expert systems for configuration at Digital: XCON and beyond. *Comm. ACM*, 32(3):298–318, 1989.

Yves Caseau and Jean-Francois Puget. Constraints on order-sorted domains. In *Proc. ECAI '94 Workshop on Constraint Processing*, 1994.

Boi Faltings and Rainer Weigel. Constraint-based knowledge representation for configuration systems. Technical Report 94/59, LIA-DI, EPFL, Lausanne, 1994.

Felix Frayman and Sanjay Mittal. COSSACK: A constraint-based expert system for configuration tasks. In D. Sriram and R.A. Adey, editors, *Knowledge-Based Expert Systems in Engineering, Planning, and Design*, July 1987.

M. Heinrich and E.W. Jüngst. A resource-based paradigm for the configuring of technical systems from modular components. In *Proc. 7th CAIA*, pages 257–264, 1991.

Kefeng Hua, Boi Faltings, and Djamila Haroud. Dynamic constraint satisfaction in a bridge design system. In *Proceedings of the International Workshop on Expert Systems in Engineering*, pages 217–232, Vienna, September 1990. Springer-Verlag.

Rüdiger Klein, Martin Buchheit, and Werner Nutt. Configuration as model construction: The constructive problem solving approach. In *Proceedings Artificial Intelligence in Design '94*, pages 201–218. Kluwer, August 1994.

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

Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings AAAI*, pages 25–32, August 1990.

H. Simonis and T. Cornelissens. Modelling producer/consumer constraints. In *Proceedings CP '95*, Cassis, France, September 1995.

Lou Steinberg, Nashir Langrana, and Greg Fisher. MEET: Decomposition and constraint propagation in mechanical design. Technical Report WP-TR-121, Rutgers University, 1989.

Markus Stumptner and Alois Haselböck. A generative constraint formalism for configuration problems. In *3rd Congress Italian Assoc. for AI*, volume 729 of *Lecture Notes in AI*, Torino, October 1993. Springer-Verlag.

Markus Stumptner, Alois Haselböck, and Gerhard Friedrich. COCOS - a tool for constraint-based, dynamic configuration. In *Proc. 10th CAIA*, San Antonio, March 1994.

Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In Koichi Furukawa, editor, *ICLP'91 Proceedings 8th International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.

Jon R. Wright, Elia S. Weixelbaum, Karen Brown, Gregg T. Vesonder, Stephen R. Palmer, Jay I. Berman, and Harry G. Moore. A Knowledge-Based Configurator that Supports Sales, Engineering, and Manufacturing at AT&T Network Systems. In *Proc. 5th IAAI*. AAAI Press, 1993.