Towards a generic model of configuration tasks

Sanjay Mittal
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Felix Frayman
Hewlett-Packard Laboratories
3500 Deer Creek Rd.
Palo Alto, CA 94303-0867

ABSTRACT

A precise definition is provided for general configuration tasks. Two important assumptions are identified: (i) functional architecture and (ii) key component per function. A domain-independent model is presented based on these assumptions. These assumptions are shown to be both useful and tenable in real domains. They are useful because they limit the complexity of the general configuration task, determine the basic knowledge needed for solving a configuration task, and enable more efficient problem solving methods. Ideas are presented both for representing the knowledge and controlling the search. Some of these ideas were originally implemented in the Cossack expert system.

1 INTRODUCTION

Configuration is a special type of design activity, with the key feature that the artifact being designed is assembled from a set of pre-defined components that can only be connected together in certain ways. This intuitive definition fits a large number of design tasks from our everyday and professional lives. It is natural, therefore, that configuration has become an important application area for knowledge-based technologies. Starting with the landmark R1/XCON project [McDermott, 82], configuration expert systems have been built by many organizations. We have found publications describing over a dozen systems [Bennet and Lark, 1986; Birmingham et al., 1988; Bowen, 1985; Frayman and Mittal, 1987; Haugeneder et al., 1985; Lan et al. 1987; Parunak et al., 1988; Pierick, 1986; Rolston, 1986; Searls and Norton, 1988; Wu et al., 1986], not to mention many others that have been mentioned in various trade publications. Many of the reported systems configure computers, though we have also seen references to systems for configuring networks, operating systems, buildings, circuit boards, keyboards, printing presses, and trucks.

Such a wealth of experience would suggest that some understanding exists about configuration as a generic task and techniques for solving it. Unfortunately, most published accounts of configuration expert systems seem to refer to only an intuitive definition of the configuration task, and often dont describe their problem solving methods clearly enough to enable meaningful comparisons.

Furthermore, it is not clear that each of these instances of configuration tasks are similar. Can the components in each of these domains be represented the same way? Are the rules of composition similar? How are the requirements expressed? What is the relationship between the requirements and what is known about the individual components in these domains? What assumptions are made by the problem solving methods? Would a method in one system be useful in other domains? These questions need to be answered before one can develop a general theory of configuration tasks or develop a more precise taxonomy of configuration tasks.

In this paper we start by defining a general configuration task in a way that captures some of the important characteristics. We show that the general task involves exponential search in the worst case. Next, we define a restricted version of the task incorporating two important assumptions: [1], we know ahead of time the kinds of functional roles that need to be fulfilled within a configured artifact. These functional roles are often constrained by an architecture. [2], we assume that for each functional role, one or more components can be identified as a "key component", i.e., any set of components that implements that function will include one of these components.

In section 3, we analyze the essential knowledge needed by a problem solver and discuss ideas for represention. Section 4 describes a series of methods for solving configuration tasks and identifies some of the important issues. These ideas were originally developed in the context of implementing the Cossack expert system [Frayman and Mittal 1987].

2 ASPECTS OF CONFIGURATION TASKS

2.1 General definition

Making very few assumptions about the kinds of knowledge that might be available, we define a configuration task as follows:

> **Given**: (A) a fixed, pre-defined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints (B) some description of the desired configuration; and (C) possibly some criteria for making optimal selections.
>
> **Build**: One or more configurations that satisfy all the requirements, where a configuration is a set of components and a description of the connections between the components in the set, or, detect inconsistencies in the requirements.

There are three important aspects to this definition. [1], the components that can be used to design some artifact are fixed, i.e., one cannot design new components. [2], each component can be connected to certain other components in fixed and pre-defined ways, i.e., the components cannot be modified to get arbitrary connectivity. We use the idea of a "port" as an abstraction for places where a component can be connected to other components. Each component locally describes constraints on the other components that can be connected at each of its ports. For now we make no assumptions about languages for specifying such connectivities. [3], a solution not only specifies the actual components but also how to connect them together. In other words, it is not enough to just identify the components.

This definition clearly includes computer configuration problems. There one is given a fixed and pre-defined set of components (e.g., printers, displays, processors, keyboards, memory modules, operating systems), some description of the kind of computer system one wants configured (e.g., "a Xerox PC that can do word processing, accounting, can be used bilingually, prints letter quality documents, ...") and possibly some optimization criteria (e.g., should be extendable in the future, cheapest). A solution to a computer configuration problem is a set of actual components and a precise description of how they should be connected to build a computer out of these parts.

The definition we have given is general enough to cover a broad array of tasks. A customer buying a car in effect configures it from a fixed set of components (car models, engines, brakes, transmissions, etc.). Design of many single-board computer systems [Birmingham et al., 1988; Bowen, 1985] can be largely viewed as a task of selecting from a set of components such as microprocessors, memory chips, encoders, decoders, and bus drivers. Similar examples can be found in other domains.

2.2 Analysis of the general configuration task

Without making any further assumptions about what knowledge is available about components, how desired configurations are specified, or how the specifications are mapped onto components, a configuration task presents a formidable challenge. Given **N** components and **p** ports per component, the space of all possible configurations is on the order of **sqrt[(pN)!]**. For even moderate sized N and small p, the space is quite formidable, and one does not expect real configuration tasks to be solved without further knowledge.

One can actually do much better than the worst-case shown above by considering, at an early stage, the local composition constraints specified at the level of each component. Based purely on the port constraints, one can pre-compute the number of components that can connect at a port. Let **k** be the upper-bound on the number of components that can be connected at each port. Let us also assume that **S** is the upper-bound on the size of the largest configuration we are interested in. Now one can estimate the size of the space of legal configurations as follows. A generator can be built that progressively generates larger configurations upto some size S, starting with size one. Given a configuration of size k, the generator produces all configurations of size (k + 1) by extending it along any of the open ports by any of the allowable components at that port. In [Mittal and Frayman, 1989], we show that the number of possible configurations produced by the above generator is on the order of $\mathbf{O(Nk^S p^S)}$.

2.3 Restricted version of configuration tasks

We now introduce two restrictions on the general configuration task. These restrictions reduce the complexity of the task and help in identifying additional kinds of knowledge that can further reduce the complexity, at least in typical cases.

2.3.1 Functional architecture

Our first restriction is based on the following observations about design practice. Artifacts are typically (but not always) designed with some purpose in mind. Experience with designing a class of artifacts leads to an understanding of the functions that must be provided to achieve the original purpose and rules on how these functions compose and interact. Often, such functional decompositions and the accompanying constraints are codified in the form of an architecture that guides the design of such artifacts. Thus, one can talk about the "Von-Neumann architecture" (or the

stored program architecture) for computers which specifies the functions of memory as a place to store data and instructions and a processor as a device that uses a program counter to fetch the next instruction to be executed. Architectures of actual computer systems such as the IBM 370, IBM PC or DEC VAX describe in detail the functions provided by these computers and rules for composing these functions. In a similar vein, one can describe architectures for other complex artifacts such as operating systems, copiers, automobiles, or printing presses.

Our first restriction is that the artifacts are configured according to some known functional architectures. In other words, instead of trying to assemble all possible artifacts that can be created from the given set of components, one restricts the problem to those artifacts that are similar in their architecture(s). This clearly restricts the scope of the task but not in an arbitrary way.

This restriction also simplifies the task. Without knowing the functions to be achieved, one might have to compute the behavior of an arbitrary assembly of components and match it against the specifications. However, once certain architectures have been defined, it is possible to configure systems by using the architecture in a more top-down fashion. Thus, one can view configuration as a generative task. Alternatively, it can be viewed as a "recognition" or "verification" task, where the input is a particular arrangement of components, and the task is to verify that the configuration actually matches one of the architectures.

As far as we can tell, most of the implemented configuration expert systems embody some knowledge of the permissible architectures. A system such as R1/XCON which configures VAXes in effect checks that a given set of components will actually result in a valid VAX system. To do this, XCON's knowledge of legal VAX architectures is implicitly represented in its rules. Similarly, MICON only configures single-board computers that match the basic architecture defined in the knowledge base, even though it might be possible to configure other kinds of computer systems from the same set of components.

Similar assumptions are made in Cossack [Frayman and Mittal, 1987], an expert system for configuring Xerox personal computers. In Cossack, the permissible architectures of a Xerox PC are explicitly represented and it instantiates one of these architectures using a set of standard components.

2.3.2 Key components per function

Even with pre-defined functional architectures, one might have arbitrary ways of implementing the individual functions from the given set of components. This might again require a problem solver to generate arbitrary configurations and test if that configuration can indeed provide the desired functions.

However, we have noticed that in many design domains, one can identify some particular component (or a small set) that is crucial to implementing some function. For example, the printing function in a computer system crucially needs a printer component. Other components needed for the printing function such as hardware interface, data cables, power cable, fonts, and driver software can be determined once a printer has been selected. Thus, one does not need to consider arbitrary configurations for printing functions - one need only start with a printer and build suitable configurations from there. We call this a "key component" assumption. Notice, that this assumption both restricts the task (certain solutions would not be considered) and simplifies it (see sec. 2.3.3 below). Also note that this observation might help explain why functions are often given names that are the same or similar to those for their key components.

Systems such as XCON, Cossack, and MICON crucially depend on this assumption. XCON actually relies on this assumption to "infer" the functional requirements from the set of components it is given to verify. It has rules which look for certain "key" components and then ensures that other components needed to have consistent sub-systems around those components are included in the configuration. Cossack and MICON use this assumption in a generative fashion to build sub-systems which are then composed together.

2.3.3 Definition of the restricted task

These restrictions can be combined with the definition given in sec. 2.1 to arrive at an abstract definition, as follows:

> **Given**: (A1) one or more functional architectures for desired configurations, each abstractly defined by functions $\{rf_1, rf_2, \ldots rf_n; of_1, of_2, \ldots, of_m\}$, where rf_i are always needed and of_j are optional; (A2) a fixed, pre-defined set of components, where a component is minimally described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints ; (A3) a mapping from each function f_i to components c_i that are key components in providing f_i and a description of other functions that are required by c_i in order to function as f_i; (B) some description of the desired configuration, usually in the form of additional constraints on some of the f_i or the associated components; and (C) possibly some criteria for making optimal selections.
>
> **Build**: One or more configurations that satisfy all the requirements, where a configuration is a set of

components and a description of the connections between the components in the set, or, detect inconsistencies in the requirements.

These two assumptions - functional architecture and key component per function - actually dovetail quite nicely. An architecture allows one to decompose an artifact along functional lines, possibly identifying constraints that always hold regardless of how the functions are achieved. The second assumption allows this decomposition to be carried all the way to the actual physical components by identifying key components that are crucial in implementing a function along with additional constraints identified during the mapping from function to components.

These assumptions reduce each of p, k, and S that define the complexity of configuration tasks. The "key component" assumption limits both the ports (p) and the components (k) that are relevant at each port. This is because, of all the ways that a component can participate in some configuration, one is only interested in those that lead to one of the functions defined by the architecture(s). These architectural limits can be realized by restricting the representation of a component to include only those ports that participate in desired functions and by imposing additional constraints on the composability of a component. Also, by limiting the space of configurations to only those that provide certain functions, we in effect put a tighter bound on S, i.e., not all sizes of configurations can possibly constitute a valid solution.

These two assumptions also transform a tightly-coupled problem (since one can extend a partial configuration in arbitrary ways) into a more loosely-coupled problem because an architecture specifies a functional decomposition and each of the required functions can be configured somewhat independently around their key components. In a real sense, the key components act like "planning islands". Arguably, good architectures have the property that they decompose an artifact into nearly independent functions. Note that in general the problem is still coupled because of reusability of components and function sharing.

3.0 KNOWLEDGE FOR THE CONFIGURATION TASK

In the rest of this paper, we focus on this restricted version of the configuration task, though for brevity we shall not use the qualifier. In the previous section, we effectively performed a knowledge-level analysis of the configuration task. To summarize, a problem solver for the task needs three kinds of knowledge. [1], available components; [2], functional decomposition specified by the given architectures; and [3], knowledge for mapping from functions to key components. A detailed treatment is beyond the scope of this paper so we limit ourselves to a discussion of some of the important issues and describe some representation ideas.

3.1 Components

Components can be described independent of how they are used by a set of physical properties (or predicates). For example, physical properties of a printer include its dimensions and weight. Components have ports to represent "places" at which other components can be attached, e.g., typical ports for a printer include data port, power supply port, paper feeder attachment port, and a paper guide attachment port.

Ports themselves can be described in terms of some set of properties. More importantly, one can specify constraints at these ports that limit what components can be attached there. Typically, these constraints would describe properties of the components that can be connected at that port or more specifically properties of a port on another component. Thus, a printer with a "3-prong female power supply port" would have a constraint that any component that attaches to this port must have a matching "connector" port.

Finally, the sub-component relationship between components has to be explicitly represented since the available components can include two components which only differ by a particular sub-component, i.e., one already has the sub-component connected and the other does not.

3.2 Functional architecture

A functional architecture specifies a functional decomposition of the artifacts and constraints on their composition. For example, the architecture of the original IBM PC specified: [1], necessary functions such as a processor, bus, main memory, keyboard, and a booting store; [2], constraints on their composition such as, "how the booting store is accessed by the processor", "which addresses are available for main memory", "how peripheral devices are addressed by the processor", and "which addresses are set aside for the display memory"; and [3], constraints on how other optional functions may be composed with the required functions, e.g., constraints on connecting to the bus. As experience develops in connecting certain other kinds of components, one can develop a functional model of some of these options (e.g., printing, data communication, pointing device in the PC case).

Individual functions can be simply modelled by a set of properties that characterize them. For example, the printing function may be described by properties such

as speed, resolution, directionality, character and font sets.

3.3 Mapping from functions to components

Finally, we need to model the knowledge for mapping from functions to components. In general, the mapping between functions and components is many-to-many. A function can be implemented by a set of components. For example, the printing function on a PC needs as components a printer, a cable, and an interface board that connects to the bus. On the other hand, actual components are often multi-functional. For example, a motherboard on a PC contains components that can provide many functions such as the main processor, bus, memory, and data ports.

The "key component per function" assumption can be used to simplify the representation. Additional requirements for implementing a function once a key component has been selected, can be indexed via that key component. These additional requirements can be expressed as other required functions (and components directly) along with constraints on each of these.

This simplifies (but does not solve the problem entirely, also see sec. 4.2.4) the mapping between functions and components and has the added advantage that variations at the level of components can be easily accounted for because each key component can have a separate description of additional requirements. The following simple example will clarify this point. Suppose a function A can be implemented by decomposing it into sub-functions A_1 and A_2, each of which can be directly implemented by some components. Furthermore, we are given three components C_1, C_2, and C_3 such that C_1 implements both A_1 and A_2, C_2 only implements A_1, and C_3 only implements A_2. Using the key component assumption, we make C_1 and C_2 be two separate key components for A, bypassing its functional decomposition. There are no additional requirements for C_1 implementing A. However, C_2 additionally requires A_2 (or C_3 if expressed at the component level).

Notice that this augmentation of component-level knowledge to include knowledge about the functions provided and other functions needed in support of the former function, begins to blur the distinction between functions and components. One can easily imagine a continuum of concepts from purely functional concepts to actual physical components. This insight has been used by systems such as Cossack and MICON to represent components in a functional lattice.

3.4 Specification of desired configuration

Limiting the scope of the task to a given set of architectures has an important consequence for describing the input specifications. Since we are no longer interested in specifiying any arbitrary configuration that can be assembled from the available components, the input specifications simply become a way to constrain one or more of the following.

[I] Choices at the architectural level. For example, in the case of Cossack, this might involve specifying which of the possible PC architectures should be used. **[II]** Constraints on the functions allowed by an architecture. For example, a typical constraint on the processor would specify the clock rate or MIPS. Similarly, the printing function could be constrained to certain print speed or print quality. **[III]** Constraints on the actual components that are used to fulfill some functional requiement. For example, a user might simply state that they want a particular word processing program, as opposed to giving a set of functional constraints. **[IV]** Finally, the requirements might include some local or global criteria for specifying optimal configurations, e.g., cost, expandability, and compatibility in the computer domain.

4.0 PROBLEM SOLVING METHODS

Our criteria for the methods we look at will be: [1], soundness (a solution is indeed correct); [2], completeness (if a solution exists within the available knowledge it will be found); and [3], exhaustivity (all possible solutions can be found). We shall ignore issues of optimization (finding the best solution under some set of criteria). These methods represent a progression towards more efficient search. Space limitations do not permit a detailed treatment of any of the methods so we will try to cover only the essential ideas (see [Mittal and Frayman 1989a] for more details).

4.1 Bottom-up method

A simple way to solve a configuration task is with a bottom-up generate-and-test method. A generator can be built (such as the one described in sec. 2.2) that starts with one of the available components and creates viable candidates by composing other components satisfying the structural composition constraints. The test part checks a candidate against the input specifications. This method is general, sound and exhaustive, but not very efficient. The complexity of the generator directly depends on the number of components and the degree of connectivity between components (see sec. 2.2). Thus, it is viable only when there are only a small number of components and they have low connectivity.

Rest of the methods only work for the restricted version of the task and mix top-down (from function to components) and bottom-up (from components to

functions) strategies.

4.2 Mixed strategy methods

4.2.1 A Top-down method

One can build a two-stage method. In the first stage, start with the functions required by an architecture and the input specifications. For each function, select one of the key components. The key component in turn may "post" additional requirements which are in terms of additional functions, or constraints on existing functions. If a contradiction is detected, e.g., a component used to implement some function is not consistent with additional constraints on the same function, then backtrack. The second stage starts with the components identified during the previous stage, and checks that they can be consistently connected together. Failure at this stage causes backtracking to the previous stage to look for an alternative to one of the inconsistent components. Let us call this method M1. This method is also sound and exhaustive but is deficient in other ways, three of which we will cover in this paper.

4.2.2 Reducing Search

The first problem with M1 is the inefficiency of its search. There are at least two causes of this inefficiency: sequencing of functional and structural stages and thrashing.

Interleaving. M1 searches unnecessarily because the structural constraints are only checked after all the components have been identified. Thus, violation of structural constraints can be very expensive. This is because the choice of a component can recursively introduce many other constraints (and additional functions), all of which would be retracted by a contradiction. This problem can be partially fixed by interleaving stages one and two in such a way that the structural constraints are checked as early as possible, i.e., as soon as two components are identified that need to be connected together, their structural constraints are checked. Let us call this method M2.

Search ordering. The second cause of search inefficiency is thrashing caused by similar contradictions. Suppose some function A can be implemented by one of n components, only some of which can be consistently composed with components that implement another function B. Both M1 and M2 will thrash, though M1 may do much additional wasted computation also, as they repeatedly try different components for implementing the function A, failing in the same place, i.e., inconsistency with components for B. This problem can be partially solved by domain-specific heuristics, that impose an order on the sequence in which the function to component mapping is performed. Many of the implemented expert systems including XCON rely on such heuristics. A better way to handle this problem is to employ some kind of least commitment strategy that postpones making a choice as late as possible. This can improve on heuristic ordering but still suffers from the atomicity of the component as a choice.

Partial choice. An even better solution would be some kind of partial choice strategy as described by [Mittal and Frayman, 1987; Mittal and Frayman, 1989b]. The basic idea here is to enable a partial component to be selected that might represent a set of possible solutions. In our example, a partial choice for the set of components implementing function A which are inconsistent with components for B might include the description of the relevant port and structural constraints (assuming they were the same for all the offending components). A later contradiction would enable the whole set to be rejected, eliminating the thrashing. Note that a function abstraction hierarchy for components, as discussed in section 3.3, would facilitate such partial choices naturally (see the longer paper for details).

4.2.3 Re-usable components

The second set of problems with M1 and the variants discussed above deals with reusable (e.g., serially or temporally) components or component sharing. The methods described above assume that if the need for the same function is identified in two or more places during search (typically as a requirement by some component) it has to be satisfied by the same component (hence the term reusable). This is not true in general. In the computer domain, bus slots can not be shared by two circuit boards. A port cannot be usually shared, except by using some kind of switch. Memory can be shared by programs but not disk space. Part of the problem is representational, i.e., one needs to indicate whether a function (or component) that needs a component (or function) does so exclusively or not. The methods would have to be modified to handle the extra information about exclusive use. Problems still remain from the point of view of search efficiency. However, the search problems are similar to those already discussed earlier and many of the same solutions can be adapted.

4.2.4 Multi-function components

The last problem deals with multi-function components, i.e., components that can simultaneously implement more than one function. There are actually two problems here. The simpler one deals with the opportunistic use of components. Thus, if a component C_1, selected for function A_1, can also implement functions A_2 and A_3 that are identified later during problem solving, one would like our method to be able

to notice that and not select new components for A_2 and A_3. Extending M1 to handle this is not hard. In fact, Cossack did this routinely.

The harder problem is biasing the search towards preferring solutions that use minimal number of components, which multi-function components enable. While an exhaustive search method that incorporates opportunistic use of components is gauranteed to find solutions that are minimal, it might not do so soon enough for real problems where one could not afford to examine all solutions. A simple heuristic for achieving this would be to order the component choices of a function on the basis of the number of functions they implement. However, this heuristic is just as likely to be bad since it biases the search towards configurations that have multi-function components many of whose functions are not needed by the specifications. Variations on least committment and partial choice ideas might also help here since they can help in early identification of required functions. Clearly more work is needed here.

It is also possible to map a configuration problem into a hierarchical constraint problem where variables and constraints can be hierarchically nested, and introduced (or retracted) dynamically as the search progresses. One proposal for such a hierarchical constraint language has been implemented and is described elsewhere [Mittal and Davis, 1989].

ACKNOWLEDGEMENTS

We appreciate discussion and comments from Dan Bobrow, Brian Falkenhainer, Dan Russell, John McDermott, Mark Shirley, and Mark Stefik on earlier drafts of this paper. Bryan Kramer collaborated with us on the Cossack system and has helped in refining many of the ideas presented here.

REFERENCES

Bennet J.S. and Lark J.S. (1986), **Hierarchical Knowledge Systems**, US patent number 4,591,983.

Birmingham, W. P., A. Brennan, A. P. Gupta, and D. P. Sieworek [1988], **MICON: A Single Board Computer Synthesis Tool**, IEEE Circuits and Devices, Jan. 1988.

Bowen J. (1985), **Automated Configuration using Functional Reasoning Approach**, in Artificial Intelligence and its Applications, (eds.) Cohn A.G. and J.R. Thomas, John Wiley & Sons, Proceedings of Artificial Intelligence in Simulation of Behaviour, 1985, pp. 79-106.

Frayman, F., and S. Mittal [1987], **Cossack: A constraints-based expert system for configuration tasks**, in D. Sriram and R. A. Adey (eds.), Knowledge-based expert systems in engineering: Planning and Design, Sept. 1987.

Haugeneder H., Lehmann E., and Struss P. (1985), **Knowledge-Based Configuration of Operating Systems - Problem in Modeling the Domain Knowledge**, Proceedings of the GI Congress on Knowledge-Based Systems, 1985.

Lan M.S., Panos R.M., and Balban M.S. (1987), **A Knowledge-based Approach to Printing Press Configuration**, Proceedings of the Third IEEE Conference on Artificial Intelligence Applications, Orlando, Florida, 1987

McDermott J. (1982), **R1: A Rule-Based Configurer of Computer Systems**, Artificial Intelligence, Vol. 19, No. 1, September 1982.

Mittal, S., and F. Frayman (1987), **Making Partial Choices in Constraint Reasoning Problems**, Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, Washington, 1987.

Mittal, S., and H. Davis [1989], **Representing and solving hierarchical constraint problems**, SSL Technical Report, Xerox PARC, 1989.

Mittal, S., and F. Frayman [1989a], **Domain-independent representations and problem-solving methods for configuration tasks**, SSL Technical Report, Xerox PARC, 1989.

Mittal, S., and F. Frayman [1989b], **Partial choice search strategies for dynamic constraint problems**, SSL Technical Report, Xerox PARC, 1989.

Parunak, H. V. D., Kindrick J. D., and Muralidhar K. H. [1988], **MAPCon: a case study in a configuration expert system**, AIEDAM, 2(2), 1988, pp. 71-88.

Pierick J. (1986), **A Knowledge Representation Technique for Systems Dealing with Hardware Configuration**, pp. 991-995, Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, 1986.

Rolston D. (1986), **An Expert System for DPS 90 Configuration**, Proceedings of the Fifth Annual International Phoenix Conference on Computers and Communications, Scotsdale, Arizona, March 1986.

Searls D.B., and Norton L.M. (1988), **Logic-Based Configuration with Semantic Network**. UNISYS internal report. To appear in the Journal of Logic Programming.

Wu H., Chun H. W., Mimo A. (1986), **ISCS - A Tool Kit for Constructing Knowledge-Based System Configurators**, pp. 1015-1021, Proceedings of the Fifth National Conference on Artificial Intelligence, Philadelphia, 1986.