Proc. of the 1991 IEEE
Int. Conf. on Tools for AI
San Jose, CA-Nov. 1991

Knowledge-based Configuration of Computer Systems Using Hierarchical Partial Choice

Bryan M. Kramer*
Department of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4
kramer@ai.toronto.edu

Abstract

As the complexity of computer systems grows, configuration expert systems become increasingly important as tools to ensure that delivered systems are workable. This paper introduces a novel inference scheme called hierarchical partial choice that efficiently generates configurations from a knowledge base of structured descriptions of computer components. This approach combines the acquisition and maintenance advantages (over rule-based systems) of a declarative system with an inference scheme that efficiently generates solutions, often with little backtracking.

The inference scheme is presented in the context of XKEWB, a shell for building computer configuration expert system. The system contains several improvements over its predecessor Cossack. For example, the system is designed to allow configurators to configure multiple computers in a network setting, the end user interface is particularly suited to the task of specifying computer systems.

1.0 Introduction

As computer systems grow in complexity, automated configuration systems become increasingly important. Questions one might have to ask while configuring a system include: is the accounting software compatible with the word processing software, and, does the file server have enough disk space for the application? Clearly a great deal of knowledge is required to correctly configure a system, and, given the large variety of hardware and software, the potential search space is very large. Although several approaches to the problem have been described, the problems of knowledge acquisition and maintenance of configuration knowledge and efficiently generating complex configuration do not appear to have been adequately addressed. This paper describes XKEWB, a shell for building computer configuration systems. XKEWB provides an effective representation of configuration knowledge and efficient generation of correct configurations. XKEWB also provides a unique end user interface that is well suited to configuration systems.

The configuration problem solved by XKEWB is a generalization of the problem described in [7]. Given a set of descriptions of components and constraints on the ways instances of these descriptions can be connected, a valid configuration is a set of instances of the descriptions and a description of their connections that satisfies the predefined constraints and some set of input constraints. Two restrictions on the general problem are described: the *functional architecture* restriction which states that descriptions of components are decomposed along functional lines, and the *key component* restriction which states that there is some key component implementing each function. Under these restrictions, one describes a computer system as a component having a display function, a computing function etc. and that the key component for the display function is a screen. The advantages of this approach are that there is a top down organization on how components are selected: to build a workstation configuration one provides the display function, the computing function etc. and that the key components identify subproblems that can be solved somewhat independently. These restrictions are implicit in the knowledge representation described in this paper.

Interest in configuration systems dates back to the R1/XCON project [5]. Configuration systems are typically rule-based or hybrid rule and frame-based systems. The problems with rules have been well documented (for example, [2]). Maintenance and verification of knowledge expressed as rules can be very difficult because related knowledge is usually spread over several rules and changes to rules have to be checked for interactions with most other rules. On the other hand, rule-based systems have the property that heuristic knowledge for finding solutions efficiently is encoded in the rules. XKEWB uses a frame-based representation to handle the maintenance problem, and has an inference engine that generates solu-

* : This work was done while the author was with Xerox Canada Inc.

0-8186-2300-4/91 $01.00 © 1991 IEEE

tions efficiently enough that procedural guidance is not necessary.

The results described here are the result of a reimplementation and extension of the Cossack configurator described in [4]. The reimplementation contains novel solutions to several weaknesses in the Cossack inference scheme, knowledge representation, and user interface. In the course of this reimplementation, several new capabilities were introduced, particularly the ability to configure multiple computers in a network situation.

The paper is organized as follows: first, a brief review of Cossack is presented. Next, there is an overview of XKEWB followed by several sections describing details of the system. Finally, the concluding section contrasts XKEWB with other configuration systems and summarizes the contributions.

2.0 The Cossack System

In Cossack, components are represented as LOOPS objects and classes of components as LOOPS classes. Interrelationships between components or component classes are represented by slots with associated *constraints*. A constraint specifies which objects might fill the slot by either enumerating the possibilities or specifying a class whose instances might fill the slot. In addition, a constraint might have an associated LISP expression that would evaluate to true or false given a particular candidate value for the slot. Thus, a class describing a high end computer might have a slot called *printer* with a constraint specifying that the value must belong to the class *Printer* and a lisp expression that evaluates to true when the value of the *pagesperminute* slot of a chosen printer is greater than 10.

The inference takes as input some constraints specified by the user. The state of the inference is represented by a set of *goals* describing components to be selected and the set of *posted constraints* which are constraints that have not yet been processed. For each posted constraint, the system first attempts to attach the constraint to an existing goal. Thus, if there is already a goal to find a printer and a second constraint requiring a printer is encountered, the system will try to satisfy both constraints with the same goal. If there is no component matching the constraint, a new goal is created. When there are no unprocessed constraints, some goal is *executed*. Executing a goal involves selecting a component that satisfies the constraints. When the choice is made, new constraints are posted, and the cycle continues.

If no component satisfies the constraints attached to a goal, a contradiction is implied. Cossack then backtracks by undoing the choice at some goal that has posted a constraint that is attached to the failed goal. Undoing the choice at the posting goal has the effect of removing a constraint from the failed goal potentially allowing a choice to be made there.

Cossack makes use of a version of *partial choice* [6]. When there are several components that might satisfy a goal, Cossack attempts to find a constraint that these components have in common. If there is such a constraint, the constraint is posted and the goal is suspended. When a component is selected for that constraint, the system can make a choice for the suspended goal using the knowledge derived by making a choice for the constraint. This is a variant of *least commitment* in that a choice is made at random only when necessary.

3.0 XKEWB

Cossack had several weaknesses. The most important is that the backtracking mechanism frequently pruned the part of the search space containing the correct solution. Second, the partial choice mechanism was rarely invoked because of difficulty of determining that a constraint was common to a set of components. Third, it was difficult to express extra conditions on constraints using the associated lisp expression, and more importantly, the information contained therein was not available to the inference engine. Finally, the representation lacked the ability to describe components that partially used other components (such as disk space) and to distinguish components belonging to different computers in a multi-computer setting.

The XKEWB system was developed in order to address these weaknesses. At a high level, the system bears a strong resemblance to Cossack. Components are described in a frame-based language, and inference consists of a loop in which constraints are attached to goals and then a goal is executed, possibly generating new constraints. When a goal cannot be successfully executed, backtracking is necessary. For example, the input constraints might specify a computer and a word processing package, hence goals to find a computer and to find a word processing package would be created. Executing the word processing goal might result in a choice, *Microsoft_Word*[1] which has a constraint that the computer have a VGA monitor. This constraint would then be attached to the computer goal. Executing the computer goal would result in a choice which would post constraints requiring disk drives, monitors, etc.

XKEWB's representation language is more declarative and expressive than that used in Cossack. Components are

1. Microsoft Word is a trademark of Microsoft Corporation

represented using a frame representation[1] with multiple and strict inheritance. Frames representing components are called *classes* and are organized in an *IS-A* (specialization) hierarchy. The slots of a frame have several facets, and the combination of slot and facets is called a constraint and corresponds to a Cossack constraint.

Below are two partial descriptions which illustrate the component representation

```
Workstation (abstract) IS-A Computer
    subcomponents
        display: WorkstationDisplay
            number: [1, 2]
        keyboard: WorkstationKeyboard
        disk: WorkStationDisk
        memory: Memory
    requires
        networkconnection:
            NetworkConnection

Ventura_Publisher[2] (individual) IS-A
                        PublishingPackage
    requires
        printer: LaserPrintingResource
            constraint:
                printer.postscript-capability
                    = true
        disk: ExtStorResource
            level: 657
            consumes: bytesConsumed
        memory: MemoryResource
            constraint:
                memory.supplied-by =
                    workstation
        workstation: WorkStation
    properties
        price: 2000
```

Here, the class *Workstation* has a constraint described by the *display* slots that specifies that a workstation is connected to one or two instances of the class *WorkstationDisplay*. The following sections describe the major steps of the inference algorithm. The discussion will make use of the above examples, and the meaning of the various parts of component description will be clarified.

4.0 Attaching Constraints

The first step of the inference algorithm is to attach newly posted constraints to goals either by finding an existing goal that is compatible or by creating a new goal. The attaching step is important because it allows components to be described independently of other components that might require the same resources, and yet results in configurations where one component is chosen to satisfy multiple requirements. For example, the descriptions for several software packages might specify that a printer is required, and the attaching step will attempt to find one printer to satisfy all of the packages. Notice that each request for a printer could add additional conditions to a goal. Thus a word processing package could request a letter quality printer and a legal package would add the condition that the printer must be able to handle legal size paper.

While it is frequently desirable to satisfy several constraints with a single component, it is often impossible to do so. The following paragraphs describe properties of a constraint that are used to refine the description of what components can satisfy the constraint, and how these properties are used in matching constraints and goals.

First, constraints fall into one of three categories: *subcomponent* constraints, *requires* constraints, and *properties*. Subcomponent constraints describe components that are in some sense supplied by the containing component and would not be supplied by a second component. For example, a *Workstation* supplies a *display* therefore choosing a second workstation would result in a second display. Thus the inference engine will attach only one subcomponent constraint to a goal. *Requires* constraints, on the other hand, describe components that are needed but might be shared. Thirdly, properties are descriptions for which no inference is required to find the intended instance. For example, no further inference is required to find the price of *Ventura_Publisher*. Properties generally describe objects such as numbers, booleans, or strings that are not components.

Another important facet of a slot is the *constraint* expression. The constraint expression of a constraint *C* constrains the slot values of objects that can satisfy *C*. Thus, the *printer* constraint in *Ventura_Publisher* can only be satisfied by printers that have the value *true* for the property *postscript-capability*. The constraint expression may involve other slot values as in the case of the *memory* slot in *Ventura_Publisher*. Here the value of the *supplied-by* slot of the memory must equal the value of the *workstation* slot of the publishing package. This constraint expresses the fact that the memory requirement is on the computer on which the publishing package is to run; not on some other computer in the configuration. Before attaching a constraint, the inference engine makes a simple analysis of the constraint expression in which it checks for equalities that might contradict slot values already bound in the goal.

A constraint expression is a boolean expression using the logical connectives *not*, *and* and *or*, comparison opera-

1. for example, FRL [8] which introduced the terms frame, slot, and facet
2. Ventura Publisher is a trademark of Ventura Software Inc., prices and storage levels are fictitious

tors such as > and =, predicates such as *instance-of*, and expressions using an assortment of arithmetic and other operators.

For the *display* slot, an interval is specified for the *number* facet that specifies that a workstation has from one to two displays. This is in effect two constraints. XKEWB assumes that the values in a multiple-valued slot are intended to be different will never attach the corresponding constraints to the same goal.

The *disk* constraint is an example of a *resource consumption* constraint. The value of the *consumes* facet identifies a slot in the type, *ExtStorResource*, and the facet *level* specifies a value. The meaning of this constraint is that the value of the slot *bytesConsumed* in an instance of *ExtStorResource* is equal to the sum of the levels consumed by all resource constraints that have that instance as a value. The following is a possible definition of the resource:

```
ExtStorResource (abstract) IS-A Resource
    levels
        bytesConsumed: Number

ExtStorResource1 (individual) IS-A
                              ExtStorResource
    requires
        diskDrive: DiskDrive
            constraint: diskDrive.capacity >
                                  bytesConsumed

ExtStorResource2 (individual) IS-A ExtStorRe-
    source
    requires
        diskDrive1: DiskDrive
        diskDrive2: DiskDrive
            constraint: diskDrive2.capacity >
                (bytesConsumed -
                          diskDrive1.capacity)
```

The specializations of *ExtStorResource* describe two different implementations of the resource, each constrained to supply at least as much disk space as is consumed. This allows XKEWB to configure the external storage as either one or two disk drives. In addition, the level slot of a resource can have a *maximum* facet which restricts the sum of the levels used by the constraints sharing that resource.

The *use* facet of a slot is also used to control when a constraint is attached to a goal. If the *use* of a constraint attached to a goal has the value *exclusive*, no other *requires* constraint may be attached to that goal. An example is a printer cable that makes exclusive use of the ports to which it will be attached.

The final attachment mechanism is a context mechanism which restricts sharing of components. Every constraint is in a context that is generally inherited from the object containing that constraint. However, by specifying a *container* facet for a constraint, one can indicate a larger context for a constraint. For example, the software selected for a particular workstation would have constraints that could share components in the context established by that workstation. One might, however, want to specify the *network* as the context for the printing requirement so that the printer is not constrained to be a subcomponent of the workstation. The complement to the *container* facet is the *setContainer* facet which indicates that the selected component is to establish a context of a certain type.

5.0 Executing a Goal

The properties of a goal are its attached constraints, its current choice, and its posted constraints. The current choice is a class in the knowledge base that satisfies all of

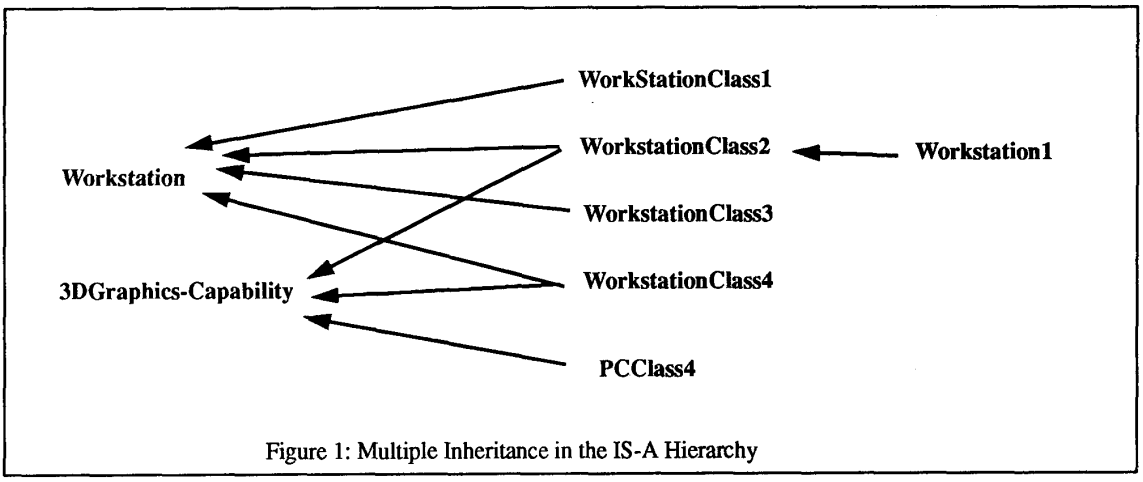

Figure 1: Multiple Inheritance in the IS-A Hierarchy

the attached and posted constraints. Classes in the knowledge base are labelled either *abstract* or *individual*. When the choice for a goal is an *individual*, that goal is satisfied. For example, a choice of *WorkStation* is abstract and requires further refinement, while a choice such as *COMPAQ_DESKPRO_386N*[1], if it is an individual, would mean that the goal is satisfied.

In order to deal with multiple inheritance, executing a goal involves choosing a description from the *most general common specializations* of the current choice and the types of all attached constraints. The *most general common specializations* of a set of classes C to be a set S of classes such that each element of S is a specialization (IS-A descendant) of every class in C, and no element of S is a specialization of any other element of S. For an example of the use of multiple IS-A parents, consider a class *Workstation* that is specialized into *WorkstationClass1*, *WorkstationClass2*, etc. Suppose also that some of these workstations have a 3-D graphics capability which is described by several constraints. Rather than duplicate the definition of these constraints in each workstation which has it, one can define a class *3DGraphcs-Capability* and make those workstations that have the capability IS-A descendants of this class. This is illustrated in figure 1 where *WorkstationClass2* and *WorkstationClass4* are the most general common specializations of *Workstation* and *3DGraphcs-Capability*, while *Workstation1* is not since another common specialization subsumes it.

From the most general common specializations, a class is a valid choice if it satisfies the constraint expressions of all attached and posted constraints. For example, the goal for *printer* would be satisfied by *LaserPrinter* if an attached constraint specified *print-quality* = *letter*. Posted constraints are often relevant as well. For example, for the choice *Printer*, the goal might have posted a constraint for an attached computer which has been specialized to an individual computer, say *ModelX*. This constraint might have the expression *port-type* = *computer.port-type* which specifies that the computer's port type must be compatible with the printer's. Clearly, a the printer class with port type *centronics* cannot be chosen if the computer's type is *serial*. Note, that when the choices are not individuals, there will be slots that are not bound, so evaluation of the constraint expression will result in *unknown*. In this case, the class is allowed as a choice.

Making a choice from the most general common specializations of the current choice and attached constraints has the effect that constraints common to a group of components are tested before each of those components is tried. That is, the algorithm uses the structure of the component hierarchy to determine common constraints. Whereas Cossack searched for common constraints from among all individual candidates for the goal, XKEWB is able to simply pick them from the current choice. For example, the class *GraphicsSoftware* might have subclasses *HighEndGraphics* and *LowEndGraphics* where *HighEndGraphics* might have a constraint that requires a high resolution display. Rather than trying each high end graphic package only to fail on the display requirement, XKEWB first specializes *GraphicsSoftware* to *HighEndGraphics* and posts the requirement. If the constraint cannot be satisfied, backtracking will result in trying *LowEndGraphics*. This technique, which will be called *hierarchical partial choice* can result in a great deal of pruning of the search space.

As well as the early detection of contradictions, hierarchical partial choice can limit the search space by providing information that restricts the choices at a goal. For example, the goal to select a workstation might post a requirement for a cpu chip. The goal selection heuristics (discussed below) would favour specializing the cpu chip goal before the workstation goal, hence when it becomes time to select a workstation, the cpu chip will be more tightly constrained, say to *80386*[2]. Thus only specializations of *Workstation* that are compatible with this choice need be considered. This is very important since the choice of cpu chip might have been constrained by the user either directly through an input constraint, or indirectly through a constraint on some other aspect of the system such as a software package.

Once a choice has been made, any new constraints are posted. The new constraints are all constraints defined in the choice and all of its IS-A ancestors that are not already posted constraints of the goal. The new constraints are put into the list *NewConstraints*, and the algorithm returns to the constraint processing step.

6.0 Backtracking, Goal Selection, and Preferences

Should there be no valid choice, it is necessary to backtrack. For the purposes of backtracking, the configuration process is a sequence made up of the operations "attach a constraint to a goal" and "make a choice for a goal". If the previous step was "attach a constraint to a goal", backtracking involves attempting to make a new attachment. Note that creating a goal counts as an attachment and can only be tried once for a constraint. If a new attachment is not possible, backtracking is again necessary. Similarly, if the previous operation was "make a choice for a goal", a

1. COMPAQ and DESKPRO are registered trademarks of Compaq Computer Corporation

2. 8086, 80286, 80386 are trademarks of Intel Corporation

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.