Journal of Intelligent Manufacturing (1994) **5**, 33–45

DIDS: rapidly prototyping configuration design systems

ALAN BALKANY, WILLIAM P. BIRMINGHAM,‡ BRUCE MAXIM, JAY T. RUNKEL and IRIS D. TOMMELEIN*

Electrical Engineering and Computer Science Department, Civil and Environmental Engineering Department,‡ Computer and Information Science Department (Dearborn), The University of Michigan, Ann Arbor, MI 48109, USA*

Received March 1992 and accepted January 1993

The domain independent design system (DIDS) provides a set of tools for rapidly constructing new configuration design systems from a library of reusable software elements called *mechanisms*. A DIDS user begins by creating a model of the problem domain and the task to be automated. This includes describing a library of parts from which new artifacts could be configured, optimization and preference criteria, and functionality constraints. DIDS analyzes this input and automatically builds an operational prototype system by selecting and combining mechanisms. DIDS' ability to automate this process is derived from its model of configuration design, which enables reusable mechanisms to be identified and automatically selected based on a problem's characteristics. The use of DIDS is illustrated by showing how DIDS solved an elevator-configuration problem.

Keywords: Design, knowledge systems, configuration, mechanisms, problem-solving methods, knowledge acquisition, DIDS, UT

1. Introduction

Knowledge-based design systems such as R1 (also known as XCON) (McDermott, 1980), M1 (Birmingham *et al.*, 1992), VT (Marcus *et al.*, 1987) and SightPlan (Tommelein *et al.*, 1992) have demonstrated the benefits of automating design tasks where essentially the same artifact is repeatedly designed with only minor variations, i.e. specifications vary slightly from design to design. Variations in design will depend on input specifications and the selection of parts available in a particular manufacturing process. Design systems should therefore take into account manufacturing constraints and process limitations.

Knowledge systems (KS) have significantly reduced both the cost and time required to produce a design by modeling the problem-solving process of humans and using KS techniques for implementation. In addition, their systematic encoding of the design process has reduced the number of errors in a design, and given a large number of organizations access to the design expertise encoded within them (e.g. Barker and O'Conner, 1989).

Although the benefits of these design systems are great, the cost and level of expertise required to build them impedes their development for new applications. A development team, consisting of both domain experts and knowledge engineers, typically requires several person-years to build a system. Designers require the assistance of knowledge engineers because they are not versed in artificial intelligence concepts and techniques. The long development time results from two factors. First, designers must communicate all relevant domain concepts to knowledge engineers, which is a slow and iterative process. Second, design systems tend to be large programs that are difficult to construct (Boehm, 1987). By automating parts of this process, the domain-independent design system (DIDS) (Runkel *et al.*, 1992) reduces the amount of time and expertise required for such development.

The DIDS model facilitates the rapid development of systems that perform a restricted form of design called *configuration* (Mittal and Frayman, 1989). Knowing characteristics of this restricted task, and when given domain and problem descriptions, the DIDS system automatically configures a design *problem-solving method* (PSM) (McDermott, 1988) from a library of reusable software elements, called *mechanisms*. The resulting system can then create designs. DIDS thus enables users with a limited knowledge of artificial-intelligence techniques to build systems, because most of these techniques are encoded in mechanisms and are, therefore, hidden from the user.

The success of DIDS depends on the development of a mechanism library which has a manageable size, but which provides enough elements to cover a significant percentage of configuration tasks. The authors' research (Balkany *et al.*, 1993) and the design literature suggest that a library of reusable mechanisms can be created. Chandrasekaran (1986) and Tong (1987) have identified fundamental elements of the design process that are shared across domains. They have proposed models of design that place structure on the design process, while implicitly assuming commonality between design processes in different domains. Others have demonstrated that design tools intended for one domain can be ported to new domains (Langrana *et al.*, 1986; Brown and Chandrasekaran, 1987; Maher, 1987; Johnson and Hayes-Roth, 1988; Birmingham and Tommelein, 1992). These observations laid the foundation for the development of DIDS.

This paper is organized as follows. The next section discusses the DIDS model of the configuration task. Section 3 describes how design tools are built with DIDS. Section 4 describes the DIDS mechanism library, and Section 5 provides an example of how DIDS can be used to create a knowledge system that solves an elevator-configuration design problem. Section 6 presents research related to DIDS and Section 7 summarizes the paper.

2. The DIDS model of configuration design

2.1. *Configuration design*

The authors' research has emphasized the automation of configuration–design tasks. Configuration design can be characterized by the following (adapted from Mittal and Frayman, 1989):

> A designer constructs an artifact given a fixed library of parts, a set of constraints relating the functionality and characteristics of these parts, specifications on the artifact's functionality, performance, and cost. The artifact must obey either rules of interconnection, geometry, topology, or any combination of the three.
> Optionally, a set of preference or optimization criteria can be given. The artifact conforms to these criteria.

A model of configuration design forms the foundation of the DIDS approach. The model enables DIDS to provide support for automating KS development by allowing DIDS to make inferences using the model's assumptions. The model also establishes a set of principles that guide the identification of mechanisms. It identifies the types of knowledge that must be acquired from the user, the mechanisms required to automate configuration tasks and the knowledge-acquisition procedures necessary to build knowledge-acquisition tools. In addition, the model identifies, for each knowledge type, mechanism and knowledge-acquisition procedure, the features of the task that indicate when it should be used.

2.2. *Mechanisms and problem-solving methods*

The DIDS model, as well as the system, is based on the concept of a mechanism. For configuration design, mechanisms represent the various techniques for automating the configuration subtasks. Two mechanisms may automate the same subtask (e.g. part selection or arrangement), but will differ by the algorithm used or the types of knowledge used. The model associates with each mechanism a collection of programming-language statements that implement the mechanism, a set of mechanism-selection features that describe when the mechanism should be used, a procedure for acquiring the domain knowledge required for the mechanism to operate and a description of the mechanism's inputs and outputs. For example, Fig. 1 shows the pseudo-code and inputs and outputs for a mechanism that performs part selection. Examples of the other types of information associated with mechanisms will be shown later in this section.

DIDS-generated problem solvers consist of a sequence of mechanisms called a problem-solving method (PSM), which is the series of steps used by the system to automate the configuration task. For example, Fig. 2 shows a PSM generated by DIDS to automate the task performed by VT, an elevator design system (Marcus *et al.*, 1987). The mechanisms are highlighted in bold, and the outermost WHILE loop defines the loop over which the problem solver iterates. Mechanisms that execute conditionally are contained within the IF and WHILE statements inside the outer WHILE loop.

select-part mechanism
Input: Abstract parts, list of possible parts, applicable constraints
Output: A part, from part list input, that implements the function input without violating any constraints.
Returns: TRUE if a part was selected.
Pseudo-code:

```
For each part in the part list
    test to see if it violates any constraints
        if part does not violate any constraints,
            then select it and return TRUE
If all parts violate a constraint,
    then return FALSE.
```

Fig. 1. A mechanism for selecting parts.

```
initialize_taskqueue(readytasks);           //Puts the first part into the queue
WHILE (not_empty_queue(readytasks))
{
    // applies compute_spec_values mechanism to each part in the queue
    apply_to_queue(compute_spec_values, readytasks);
    get_next_task(readytasks, action);      //Gets the first part in the queue
    add_to_design(ds, action);              //Adds the part to the design
    IF (is_an_and_node(action))
    {
        select_all_parts(action, candidates);       //selects all of action's children
        add_tasks(readytasks, candidates);          //and adds them to the queue
    }
    ELSE IF (is_an_or_node(action))
    {
        //selects the subset of actions children that satisfy the constraints
        select_candidate_parts(action, candidates);
        select_best_part(candidates, newpart);      //selects the part with the least cost
        add_task(readytasks, newpart);              //adds the part to the queue
    };
}
```

Fig. 2. A problem-solving method.

2.3. *Knowledge types*

The DIDS model defines the types of knowledge required to perform configuration design. These types represent different classes of concepts, which are necessary to automate configuration and are used to form a conceptual model of the problem domain. They were identified by studying existing configuration design systems (Balkany *et al.*, 1992), and the authors believe that they are sufficient to represent all the domain knowledge necessary for configuration tasks.

A knowledge base in the DIDS model is a graph, where the nodes correspond to domain concepts, and links represent relationships between them, as is shown in Fig. 3. The knowledge types define the different types of nodes in this graph and the possible relationships between them. Only two knowledge types form nodes in this graph: 'part' and 'abstract part'. These two types represent parts that can be used to build an artifact and the functions that the artifact must perform, respectively.

DIDS-generated systems produce designs by successively decomposing the abstract parts into lower-level abstract parts, and by selecting parts to implement

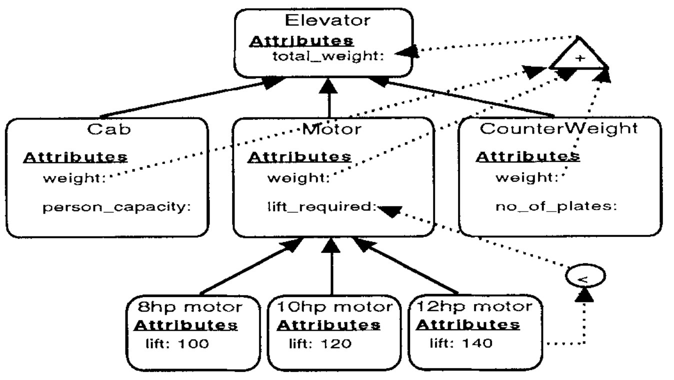

Fig. 3. A few of the knowledge types in the VT knowledge base.

low-level functions. The remaining types, which define the relationships between parts and abstract parts, i.e. links in the graph, represent knowledge that is used to guide the search for the best abstract part decompositions, and the best set of parts that can be used to implement the abstract parts.

The nine knowledge types are the following:

(1) *Parts* – the part knowledge type represents the elements in the part library. Parts are defined by a set of characteristics, ports and boundaries. Characteristics define the properties of a part that can be expressed by a name and a value. The values of characteristics are defined before problem solving begins and cannot change during problem solving. The ports of a part define where it can be connected to other parts. The boundaries of a part define how the part can be arranged relative to other parts.

(2) *Abstract parts* – abstract parts represent the functions and subfunctions that the artifact being designed must perform. Abstract parts are defined by their characteristics, ports, boundaries and specifications. Specification values depend upon the design problem being solved, and therefore their values must be computed during problem solving.

(3) *Subfunction* – the subfunction knowledge type successively decomposes the artifact being designed along functional lines. It describes the functional relationship between the parts and abstract parts in the domain. This relationship describes how abstract parts may be realized by combining sets of lower-level functions, which may include parts.

(4) *Required functions* – parts and abstract parts often require the functions performed by other parts to support their operation. This information is contained in the required-functions knowledge type. Associated with each function performed by a part is a list of required functions to support its operation, but which are not specified by the user.

(5) *Attribute constraints* – attribute constraints specify algebraic relationships between the attributes of parts and abstract parts that must be maintained. Constraints enable the problem solver to distinguish acceptable from unacceptable solutions, and to compute specification values.

(6) *Connection constraints* – connection constraints restrict the set of possible connections that can be made among the ports of parts and abstract parts. They may either specify illegal connections, or sets of connections that have been found to be useful in the past.

(7) *Arrangement constraints* – arrangement constraints restrict how parts can be geometrically or topologically arranged. They define the (physical) relationships between the boundaries of parts and abstract parts.

(8) *Preference knowledge* – preference knowledge

enables a design system to choose between sets of acceptable design alternatives. Preferences differ from constraints in that constraints eliminate alternatives, while preferences rank a set of acceptable alternatives so that optimal designs can be found.

(9) *Task-ordering knowledge* – task-ordering knowledge describes the most efficient order in which to tackle subtasks.

The knowledge types define the types of domain knowledge that must be acquired from a domain expert. In addition, they act as a set of primitives, which are used to define the functionality of mechanisms, the knowledge communicated between mechanisms and the types of domain knowledge used by the mechanisms. All mechanisms are defined by the operations that they perform on the knowledge types; i.e. the inputs and outputs of mechanisms are knowledge types. For example, a mechanism may take an abstract part as input and output the part that can be used to implement it, or be given a set of parts and determine the connections between them.

Defining mechanisms in this way has two advantages. First, the mechanisms will be reusable and combinable. The definition ensures reusability, because it places no restrictions on the specific domain concepts that must be supplied, or on the source domain of the concepts. The only requirement is that the knowledge-type classification of domain concepts is the same as the inputs to the mechanism. This guarantees that a mechanism can be applied to any configuration task where the domain contains the appropriate knowledge types. The knowledge-type definition also ensures the combinability of mechanisms, since all mechanisms share a common representation of these types. Therefore, any two mechanisms that use the same types of knowledge can share information, and can be easily combined. Second, this definition makes it clear exactly what knowledge must be in the knowledge base for each mechanism to operate. This information can be used to guide the selection of mechanisms when constructing a problem-solving method, and to guide the construction of a knowledge-acquisition tool for the method.

The knowledge types provide a significant portion of the information used by DIDS to select mechanisms. Most of the information necessary to make mechanism selections can be determined by analyzing the organization of knowledge types in the domain. Selections are guided by the absence or presence of knowledge types, and by analyzing the relationships between types. The possible variations in the knowledge types are called *mechanism-selection features*, and are enumerated in the model. For example, Fig. 4 shows the single/multi-function selection feature, which measures the possible variations in abstract-part decompositions. In the first decomposition, each (abstract) part is a subfunction of exactly one other abstract part; it forms a tree. In the second decomposition, each (abstract) part may be the subfunction of more than one abstract part; it remains a tree. For the first decomposition, a simple part-selection mechanism can be used, but for the second one a more sophisticated mechanism, such as GOPS (Haworth *et al.*, 1992) must be used.

2.4. *Knowledge acquisition*

As mentioned previously, the DIDS model associates with each mechanism a procedure for acquiring the knowledge used by that mechanism. These procedures, which are called **Me**chanisms for **K**nowledge **A**cquisition (MeKA), define a model-based knowledge-acquisition tool for acquiring the knowledge types used by a mechanism (Runkel and Birmingham, 1992). MeKAs are model based because they use the mechanism's assumptions concerning the types of knowledge types available in the domain and the relationships between these knowledge types to guide knowledge acquisition. For example, if a mechanism assumes that the domain will contain an abstract part decomposition as in the example on the left in Fig. 4, then the MeKA will ensure that all abstract parts are the subfunction of, at most, one abstract part. This MeKA is shown in Fig. 5.

A MeKA has four components (Fig. 5) – infer, present, acquire and verify – which correspond to the four-step process used to acquire knowledge for a mechanism. The *infer* component uses a MeKA-specific inference procedure to automatically derive the necessary knowledge. The *present* component of a MeKA acts as a filter, presenting only the relevant elements of the knowledge base to the domain expert. The information displayed provides enough details to give the domain expert the appropriate context for the knowledge being requested, without overwhelming the user with the

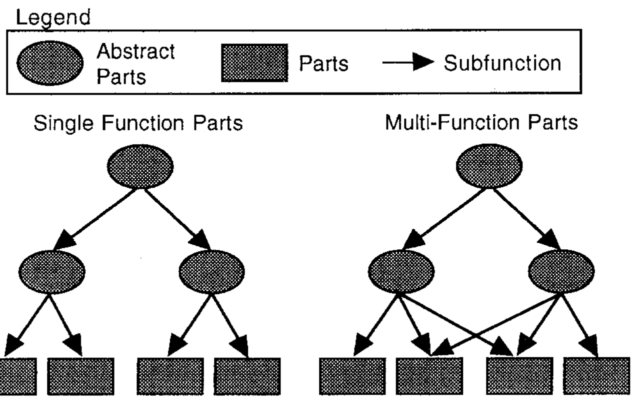

Fig. 4. Two alternative decompositions of abstract parts.

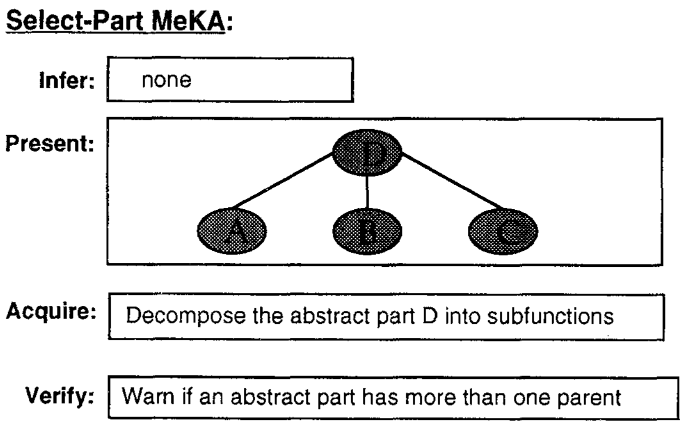
Select-Part MeKA:
Infer:
none
Present:
D
A
B
C
Acquire:
Decompose the abstract part D into subfunctions
Verify:
Warn if an abstract part has more than one parent

Fig. 5. A MeKA for a mechanism that selects parts.

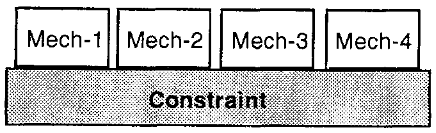

Fig. 6. Relationship between the constraint network and mechanisms.

complexities of the knowledge base. The *acquire* component either asks the user to modify the display of the present component, or asks the user about the portion of the knowledge base displayed by the present component.

2.5. *Constraint network*

The last component of the DIDS model, which is called the constraint network, provides a set of knowledge-representation constructs and inference techniques to augment the mechanisms. The network performs propagation and consistency checks over dynamic constraints (constraints that are only applicable under certain conditions). The network contains a data structure for each of the knowledge types and makes inferences using these structures. Mechanisms are built assuming the network (Fig. 6): they retrieve domain knowledge by querying the network, use the network inferences to produce their results and record their results back into the network.

When mechanisms make changes to the network, the network propagates the effects of these changes and ensures that none of the constraints are violated. The network performs a special form of arc consistency (Mackworth, 1977), called *constraint propagation* (Sussman and Steele, 1980), as the attributes of parts and abstract parts are assigned values. It uses the constraints and the known values of the attributes to compute values of attributes whose values are not known. For example, Fig. 3 shows a simple constraint between the total_weight attribute of the elevator abstract part and weight attributes of three other abstract parts: motor, cab and counterweight. The constraint states that the total_weight attribute is equal to the sum of the weights of the motor, cab and counterweight. Whenever three of the cost attributes are assigned a value, the network automatically computes the value of the fourth.

3. Building systems with DIDS

DIDS automates the KS-development process through the use of a set of tools based upon its model of configuration design. These tools assist with development by helping a user to rapidly combine the model elements. In this way, DIDS reduces both the amount of time and the level of expertise required to build systems. DIDS begins by presenting a task-modeling interface to the user. The user uses this tool to describe a prototypical portion of the problem domain in terms of the knowledge types. DIDS then analyzes this prototypical domain description to determine the mechanism selection features, which imply the set of mechanisms and MeKAs required. DIDS combines the mechanisms to form a PSM for the user's task and a knowledge-acquisition tool.

It is not expected that DIDS will generate the correct PSM and knowledge-acquisition tool on the first attempt. Often a user's prototypical domain description may be naive or incorrect, leading to incorrect mechanism selections. Therefore, the DIDS development process is iterative. When users uncover problems with the PSM or knowledge-acquisition tool, they revisit the initial task-modeling interface to modify the domain description. DIDS then generates both a new PSM and a new knowledge-acquisition tool. The existing knowledge base, however, is not lost because of DIDS' standard representation of the knowledge types. Any knowledge acquired by previous versions of the knowledge-acquisition tool is automatically consistent with the knowledge base generated by the newest version of the knowledge-acquisition tool. This feature makes DIDS-generated systems easy to extend and to maintain.

When the needs of a system change, users must simply revisit the task-modeling tool (discussed below) to describe how the task has changed. DIDS then automatically generates a new PSM and knowledge-acquisition tool to reflect these changes. The new knowledge-acquisition tool acquires any knowledge required to perform the new version of the task that is not already contained in the knowledge base. For example, consider a DIDS-generated system that selects the set of counterweights required to meet functionality specified by the user. Now, in order to extend this design system to determine the connections between the counterweights, the user visits the task-modeling tool, and describes the types of connection knowledge present in the domain. The addition of connection knowledge results in a new set of mechanism-selection features, which cause DIDS to add mechanisms to the PSM for connecting parts. The new knowledge-acquisition tool will acquire the necessary connection knowledge from the user.

The DIDS system consists of five components that automate the process outlined above: a task modeler, a mechanism manager, a mechanism library, a code generator and a knowledge-acquisition-tool generator (Fig. 7). The functionality of each of these tools is presented below. It is expected that DIDS users will visit these tools in the order listed, but since the process is likely to be iterative, this may not always be the case.

Task-modeler – the task modeler presents to the user a generic knowledge editor for describing the problem to be automated in terms of the knowledge types. Unlike model-based knowledge-acquisition tools, this tool provides very little support to the user since the system does not understand, at this point, the problem domain. It simply provides interfaces that allow the user to describe some prototypical portion of the knowledge base. The modeler analyzes the knowledge entered to determine the mechanism-selection features. DIDS questions the user directly when these features cannot be inferred. Once the features have been determined, they are communicated to the mechanism manager.

Mechanism manager – the mechanism manager selects the mechanisms and MeKAs required to automate the user's task based upon the selection features identified by the modeler. To facilitate this selection process, each mechanism in the library is annotated with the set of features that determine when the mechanism should be selected (see Fig. 12). The selected mechanisms are sequenced and connected to form a PSM using a schema that is also retrieved by matching features. The MeKAs are sequenced to build a knowledge-acquisition tool by using a set of heuristics.

Generators – the code generator and the knowledge-acquisition-tool generator take the PSM description and the knowledge-acquisition tool description produced by the mechanism manager and use them to generate the source code for the PSM and the knowledge-acquisition tool. The generators retrieve the code fragments describing the implementations of both the MeKAs and the mechanisms from the mechanism library, and combine them according to the mechanism manager's descriptions.

Knowledge-acquisition tool – the generated knowledge-acquisition tool interviews the user to build a knowledge base for the PSM. It uses the task model produced by the task modeler to provide active assistance during the acquisition process. Its assistance includes completeness and consistency checks, filtered presentations that display only the relevant portions of the knowledge bases, and an agenda mechanism that controls the order in which knowledge is acquired.

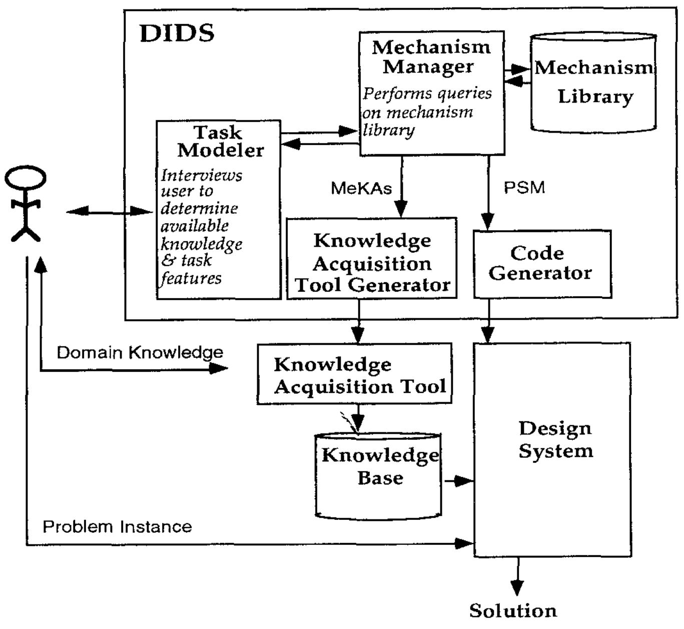
DIDS
Mechanism Manager
Performs queries on mechanism library
Mechanism Library
Task Modeler
Interviews user to determine available knowledge & task features
MeKAs
PSM
Knowledge Acquisition Tool Generator
Code Generator
Domain Knowledge
Knowledge Acquisition Tool
Knowledge Base
Design System
Problem Instance
Solution

Fig. 7. DIDS and a mechanism library.

Design system – the design system, when combined with the knowledge base acquired by the knowledge-acquisition tool, is a fully operational, domain-specific knowledge design tool. It presents an interface that allows a user to enter a problem instance for the system to solve. The system solves the problem instance by invoking the mechanisms and using the knowledge in the knowledge base.

4. DIDS' mechanism library

The heart of DIDS is the mechanism library. The library must provide sufficient coverage to allow all the configuration-design subtasks to be solved, yet its size cannot be so large as to be unmanageable. The DIDS model, specifically the knowledge types and constraint network, restricts the size of the library while supporting reusability. In this section, we show the organization of the library and provide some examples of mechanisms.

4.1. *Mechanisms*

Several of the mechanisms and their corresponding MeKAs in the DIDS system are illustrated in Fig. 8. Note that the inputs to a mechanism are knowledge types; a pointer to a specific node in the knowledge base is also provided to provide context for the mechanism. In addition, a set of mechanism-selection features is given

Mechanism	**MeKA**
compute_spec_values(Node part) {For each attribute, att, of part if att does not have a value then compute att's value using its formula } Task-Selection Features: parts have attributes Functional Group: Design extension	compute_spec_values-MeKA(*focus*) Knowledge used: Attributes Formulas Inference: none Prompt: Prompt("Use the following table to define the attributes of %s. For each attribute define either a value or a formula that can be used to compute the value of the attribute." get_name(focus)); Verify: {for all attributes, att, of focus ensure that att either has a value or att has a formula }

select_all_parts(Node nd, Set& candidates)
{set candidates equal to nd's children in the functional hierarchy.}
 	
Task-Selection Features: the hierarchy must contain AND nodes
 	
Functional Group: Design extension
	select-best-part-MeKA(*focus*)
	{
	} */* no knowledge required */*

select_candidate_parts(Node nd, Candidates set, Constraints con)
{Get *nd's* children. in the functional hierarchy;
    For each of child, *ch*, of nd
        if ch satisfies the constraints *con* on *nd* then add *ch* to candidates;
}
 	
Task-Selection Features: must have hierarchy
 	
Functional Group: Design extension
	select-candidate-parts-MeKA(Node *focus*)
	Knowledge used: Constraints
	Present:
	    present_attributes(focus);
	    present_attributes(children(focus));
	Prompt:
	    prompt("Enter the constraint used to select the part %s set.", children(focus));
	Verify:
	{The constraint acquired constrains the focus and its children.}

select_best_part(Set& cand_parts, Node& best) {set best equal to the node in the set, cand_parts, that has the cost attribute with the smallest value. } Task-Selection Features: no restrictions Functional Group: Design extension	select-best-part-MeKA(Node *focus*) Knowledge acquired: Attribute: "Cost" Inferences: none Prompt: prompt("Enter the cost in dollars of the part %s?", get_name(focus)); verify: {cost is greater than 0};

Fig. 8. A few of the MeKAs and mechanisms in the DIDS library.

for each mechanism. As more mechanisms are added to the library, it is possible that these features will be updated so that new mechanisms can be discriminated from existing ones. Finally, the mechanism is tagged with a *functional group* (described in the next section).

MeKAs are also shown in Fig. 8. Each MeKA specifies the knowledge to be acquired, the method for acquiring it (the prompt field), whether the knowledge can be inferred from knowledge already acquired (the inference field) and a method for verifying it. This is all the information needed to construct a model-based knowledge-acquisition tool.

4.2. *Functional organization of the mechanism library*

Mechanisms can be selected from the library based on task features. Guidance, however, for assembling the mechanisms into a usable PSM is required. Simply matching inputs and outputs is insufficient, since this does not consider any notion of function. For example, a mechanism that adds numbers and one that subtracts numbers will have the same inputs and outputs.

In studying several configuration tools, a PSM for configuration design has been derived (Balkany *et al.*, 1992), and is shown in Fig. 9. The PSM assumes that mechanisms can be partitioned into disjoint functional groups (select design extension mechanisms, make design extension mechanisms, select fix mechanisms and make fix mechanisms). The DIDS model ensures this, since a mechanism can only perform one operation on a knowledge type.

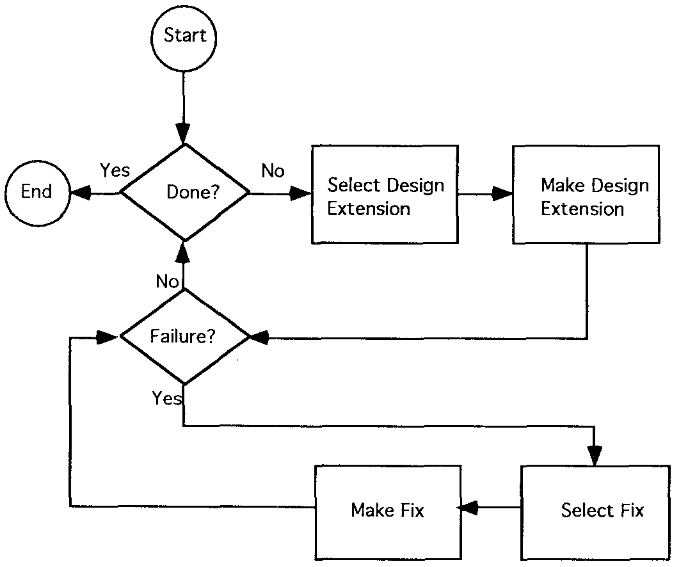

Fig. 9. The configuration-design PSM.

This model can be instantiated for a new design task by choosing the proper mechanism for each step. The control-flow relationships between different functions are already defined. In some cases, one or several steps can be eliminated. For example, some systems avoid correcting failure; hence the select fix and make fix functions can be eliminated.

5. Developing an elevator-design system using DIDS

An example shows how DIDS can be used to develop knowledge systems. The example further demonstrates the types of support that DIDS provides during the knowledge-system development process and the extendibility and maintainability of DIDS-generated systems. It shows how DIDS was used to construct a system that automates the VT (vertical transport) (Yost, personal communication) elevator-design task. Not all components of DIDS, however, have been completely implemented, to some of the functionality described in the previous sections was not available when the VT task was automated. This discussion presents the anticipated functionality of the completed DIDS system, not the functionality of the current prototype.

5.1. *VT problem*

The VT task involves designing elevators for high-rise buildings according to a set of specifications provided by an architect. The specifications describe the elevator requirements, such as the number of floors in the building, the dimensions of the shaft, the distance between each floor and the maximum capacity of the elevator. In addition, there are numerous constraints defining relationships between the elevator components that must be satisfied by the design. Designers select parts from a catalog to produce an elevator design that has low cost, meets the specifications and satisfies the constraints.

5.2. *Task modeling*

The designer begins the development process by invoking the task-modeling tool. This tool is used not to build a complete knowledge base, but to define some representative portion of the knowledge base so that mechanism-selection features can be identified. The tool provides a variety of interfaces that allow the designer to describe the problem domain using the DIDS model. It is not necessary for the designer to be aware of the intricacies of the model because the tool's interfaces allow the designer to express available knowledge using familiar notation. For elevator configuration, designers use schematic drawings of elevators to determine the necessary dimensions of its components and the forces on these components. Therefore, the designer selects the modeling tool's structure-constraint editor to draw schematics (Fig. 10).

The structure-constraint editor allows the designer to draw the elevator schematic and to label the objects and the edges in the diagram. The editor represents the schematic in the DIDS model by creating an abstract part for each object in the diagram, and an attribute on the objects for each of their labeled edges. In addition, the editor infers constraints between the attributes by analyzing the relationships between the labeled edges in the diagram. For example, the editor would create abstract parts for the cab, stile, door opening, safety and crosshead. The specifications inferred by the editor would include height specifications for the platform and safety. The editor will also create a constraint recording that the sum of the ub-space and cab-height attributes must equal the sling-ub attribute.

Once the designer has completed the schematics, the task modeler's hierarchy tool is used to further define the objects and relationships identified in the schematic diagram. The designer uses this tool to identify some of the parts in the part catalog and to relate parts in the catalog to the abstract parts that they implement. Figure 11 shows a small portion of the hierarchy created by the

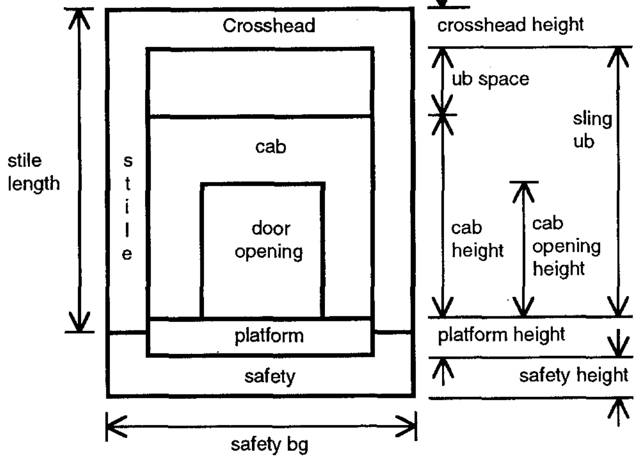

Fig. 10. Schematic drawing of elevator from Yost, personal communication.

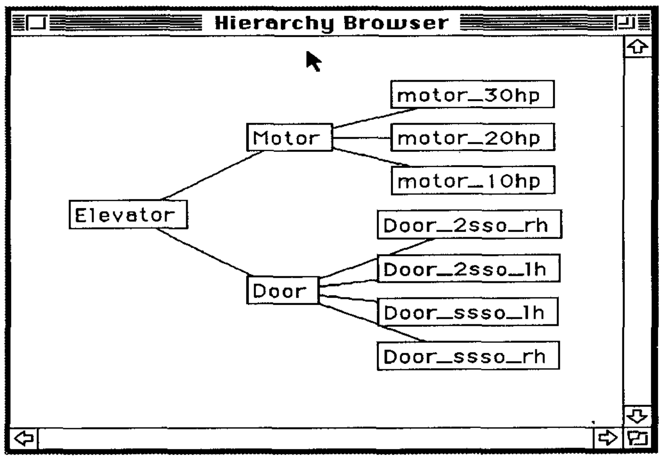

Fig. 11. Hierarchy tool showing relationship between abstract parts and parts.

designer. The abstract-part, part and constraint editors (not shown) are used to define the characteristics and specifications of each object in the diagram and the constraints between them.

The modeler also contains interfaces for acquiring preference and task-ordering knowledge. The designer enters a formula into the preference editor expressing that low-cost parts should be evaluated more favorably than expensive ones. This preference will be used by the design system to distinguish between parts that perform the same function and satisfy the specifications and the constraints. The designer does not use the task-ordering editor, indicating that the abstract parts can be designed in any order.

For the VT task, in addition to not specifying task-ordering knowledge, the designer does not use any of the

Mechanism Selection Feature	VT's Feature
functions per part	1
hierarchy levels	3
required functions	none
preference	evaluation function
task ordering	none
connection constraints	none
arrangement constraints	none

Fig. 12. Mechanism selection features determined by the modeler.

modeler's editors that allow the possible connections between parts – their legal arrangements – to be defined. This information is not necessary to configure elevators as parts can only fit together in one way, and their connections and arrangements are obvious when a schematic like that of Fig. 10 is available. Given a consistent set of parts, the workers that assemble elevators have no trouble combining them to build an elevator. The difficult task is selecting the set of parts that will form a working elevator satisfying the specifications and the constraints while minimizing cost. The design tool only needs to be concerned with the information necessary to select parts.

Upon completion of the domain description, the designer tells the modeler to generate a knowledge-acquisition tool and a design system. The modeler analyzes this description to determine the mechanism-selection features (Fig. 12). The first three features are determined by analyzing the knowledge acquired by the hierarchy tool. The functions-per-part feature measures the maximum number of parents of each part. For the VT task, each part has one parent in the hierarchy and, therefore, each part performs exactly one function. The hierarchy-levels feature measures the number of levels in the hierarchy, and the required-function feature measures whether or not the domain contains required-function knowledge. For VT, there are three levels in the hierarchy and no required functions.

The remaining features define the properties of the other knowledge types. The preference feature describes the types of preference knowledge supplied by the domain expert. As mentioned above, the domain expert supplied an evaluation function ranking low-cost parts over expensive ones. Finally, the user failed to enter any connection or arrangement constraints and task-ordering knowledge, so these features have no value.

5.3. *Generating a KS*

The mechanism manager receives the selection features from the modeling tool, and uses them to build a PSM. The manager begins with a schema, like the one shown in Fig. 9. It selects a mechanism for each functional group in the schema by finding the mechanism whose selection features most closely match those identified by the modeling tool. These mechanisms are then combined in the order specified by the schema.

The knowledge-acquisition generator builds a knowledge-acquisition tool for these mechanisms by retrieving the MeKAs for each mechanism from the library. The MeKAs are combined using a set of predetermined heuristics that order the invocation of the MeKAs according to the way designers feel comfortable describing configuration knowledge. A more detailed discussion of the knowledge-acquisition-tool generation process is given in Runkel and Birmingham (1992).

The PSM and knowledge-acquisition tool descriptions are then passed to the generators, which combine the mechanism and MeKA code segments resulting in a PSM and a knowledge-acquisition tool.

5.4. *Knowledge acquisition*

After the design tool has been generated, the designer begins the most time-consuming part of the development process: knowledge acquisition. The designer uses the knowledge-acquisition tool to build a knowledge base describing all the knowledge necessary to design elevators. The knowledge-acquisition tool, unlike the task-modeling tool, has been tailored by DIDS to acquire knowledge about elevators and provides active assistance during the acquisition process.

The knowledge-acquisition tool begins by acquiring the part library and the relationships between the parts. First, the designer must list each part in the library and define its attributes. Next, the tool invokes its first two MeKAs to present interfaces, which are similar to the modeling tool's structured-constraint editor and hierarchy tool, for defining the abstract parts and their relationships to the parts. Figure 5 depicts the MeKA used to define the relationship between abstract parts and parts. During task modeling, the designer used these interfaces to describe only a small portion of the domain knowledge. Now the designer is encouraged to be exhaustive. The designer uses these interfaces until all elevator functions have been enumerated and all parts that implement them have been defined.

The MeKAs continuously monitor the designer's domain description to ensure that it matches the mechanism-selection features. In this example, the MeKAs ensure that the hierarchy remains three levels deep, no required functions are specified and all parts perform exactly one function. This enforcement is necessary because the mechanisms were selected based on these features, and may not work properly if the knowledge base does not have them. If the designer needs to violate these restrictions, however, the task-modeling tool can simply be invoked again to make the necessary changes to the knowledge base. This will result in a new set of features and a different PSM.

Next, the add-part MeKA is invoked to acquire both the preference knowledge and the constraints necessary to select parts. The MeKA acquires a cost attribute for each part if the designer has not already supplied one and acquires, for each abstract part, a set of constraints that can be used to select among the parts that implement it. Since the add-part mechanism selects parts by looking at constraints that relate the specifications of an abstract part to the characteristics of its child parts, the MeKA acquires this type of constraint. In addition, the MeKA acquires constraints that can be used to calculate values of the specifications from the specifications of other abstract parts and the design requirements.

The knowledge-acquisition process, in contrast to the way in which it has been described here, is iterative. Often the knowledge acquired by one MeKA will cause the designer to extend or modify the knowledge acquired by a previously invoked MeKA. The designer has the option at any point during knowledge acquisition to invoke the MeKA of choice to fix some portion of the knowledge base. In addition, the knowledge-acquisition tool may invoke a MeKA when it spots an inconsistency in the knowledge base, to encourage the designer to remove it.

Once the knowledge-acquisition process is complete, the MeKAs translate the knowledge base into the constraint network. This is a simple transformation that converts the knowledge-type representation used by the knowledge-acquisition tool to the knowledge-type representation used by the mechanisms. The constraint network is then combined with the PSM to produce a KS capable of automating the VT task.

5.5 *Running the system*

The DIDS-generated system is evaluated by running it on a variety of test cases (including input specifications for a design and a designed artifact that meets those specifications, which is the expected output of the KS). Typically, these runs uncover areas where the knowledge base is incomplete or incorrect and then, the knowledge-acquisition tool is invoked to make the changes. If necessary, the designer can revisit the task modeler to make more drastic changes to the structure of the knowledge base. Once the system has been thoroughly tested, it can be used to produce working elevator designs.

5.6. *Extending the system*

The DIDS-generated design system is easy to extend and to maintain. For example, the knowledge-acquisition tool can easily be used to update the knowledge base as the parts and their functionality change over time. In addition, DIDS can be used to extend the system when the nature of the design task changes. For example, assume that a new set of elevator controller hardware becomes available. Previously, several controller modules were used, each one applicable in a different situation. Now, a few general-purpose modules can be integrated in a variety of ways. The functionality of the controller is not only determined by which modules are selected, but by how the modules are connected. The system must not only select the appropriate set of modules but determine how to connect them together.

To make this addition to the DIDS-generated system, the designer simply revisits the task-modeling tool to describe a few of the new modules, their ports and the possible connections between them. This knowledge might take the form of sets of legal module connections, or a set of connection constraints describing illegal module connections. This additional knowledge results in several new selection features, which cause the mechanism manager to add to the PSM new mechanisms that connect parts. In addition, MeKAs that acquire the connection knowledge are added to the knowledge-acquisition tool.

Since all DIDS-generated systems share the same set of knowledge types, most of the previous knowledge base can be reused. The old controller modules must be removed, the new ones defined and the knowledge necessary to connect the modules acquired by invoking the new MeKAs. The rest of the knowledge base can be reused without any modifications. Once the designer has used the knowledge-acquisition tool to make the necessary modifications to the knowledge base, the new KS is ready for testing.

6. Related work

This section compares DIDS to two classes of systems. The first class of systems automates programming in general, and the second class of systems facilitates the construction of knowledge systems.

6.1. *Automatic programming*

The programmer's apprentice (PA) (Rich and Waters, 1988), an assistant to a software engineer, facilitates program development using reusable components in all phases of the software-development process. These components, called *clichés*, represent commonly used combinations of programming elements. The PA, which contains clichés that represent familiar specification, design and implementation constructs, develops software by using inspection methods. During inspection, the PA helps the user to recognize clichés in the specifications and to choose between the lower-level clichés that implement the specification. In contrast, DIDS, which is restricted to configuration tasks, completely automates code generation by analyzing a user's prototypical domain description. In addition, DIDS uses a propose-and-revise methodology for system development whereas the PA, which has a rich representation for the behavior of clichés, supports a refinement methodology.

Draco (Neighbors, 1984) automates software development by reusing software components. Draco not only supports the reuse of code, but also the reuse of analysis and design information. Draco libraries contain domains for which the typical problem statements and implementation alternatives are known. Users develop new systems by describing requirements in terms of known domains. This allows the analyses and the designs developed for some domains to be reused on a new problem. Instead of mapping a known solution to a new problem, DIDS' mechanisms can be recombined to cover a greater range of problems. DIDS also reuses analysis and design information; in DIDS, these take the form of mechanism selection features that indicate when each mechanism should be reused.

6.2. *KS development aids*

Klinker *et al.* (1990) propose to build systems by combining mechanisms, in a way similar to DIDS. The approach differs in that their system is geared towards non-programmers, the analysis of user's tasks is an integral part of system generation and the task type is not restricted. This makes it difficult to determine *a priori* the types of knowledge and the set of mechanisms required to construct systems. Instead, they have developed a *shared vocabulary* of task activities that can be used to describe tasks in domain-independent terms. The system analyzes the user's task, and helps to describe it in terms of the shared vocabulary. Each activity in the shared vocabulary is associated with a set of mechanisms that can be used to implement it. The shared vocabulary helps to make mechanisms usable, i.e. understandable by users, and reusable.

Neches *et al.* (1991) propose to build new knowledge systems through knowledge sharing. This involves building tools that enable the knowledge base of one system to be used by another, and facilitate the communication between knowledge systems. A standard knowledge-representation system, a standard knowledge-base query language, a standard concept ontology and a standard language for expressing knowledge form the heart of this approach. DIDS has the most in common with the standard concept ontology, since the mechanism and PSM libraries can be viewed as ontologies of problem-solving components. The philisophy of DIDS, however, differs significantly from the other parts of the Neches *et al.* approach. DIDS supports the reuse of fundamental software components by identifying mechanisms shared across domains instead of reusing existing knowledge bases. We believe the task-specific bias of most knowledge bases makes their reuse difficult.

Protege II (Puerta *et al.*, 1991), a system similar to DIDS, generalizes the capabilities of Protege (Musen, 1989) and combines them with a mechanism-based model. Protege II contains a library of tasks and a library of mechanisms. A mechanism's description includes a description of the data used by the mechanism and links to tasks, which are used to suggest the mechanisms capable of performing a task. Once the mechanisms have been selected, Protege II generates a knowledge-acquisition tool by looking at the data required by each mechanism. A weaker model of expertise distinguishes Protege II from DIDS since the set of mechanisms, the types of data operated upon by mechanisms and a procedure for identifying mechanisms have not been established.

DIDS can also be compared to design programming languages. The authors' work is aimed at understanding how design systems operate, namely the way in which they solve their particular problems. What makes these systems different is the design knowledge they use, and the domains in which they operate. Thus, the authors' work is significantly different from those developing languages for constructing design systems, such as DSPL (Brown and Chandrasekaran, 1989), Edesyn (Maher, 1987) and DESCRIBE (Mittal and Araya, 1987). These languages provide programming constructs to easily capture design knowledge for specific tasks. All three, however, work at a different abstraction level than the mechanisms described in this paper. Furthermore, they provide simpler, albeit potentially more general, operators (mechanisms) than are assumed by the authors' models. In fact, the mechanisms described here could be implemented in any of these languages, as they could in more traditional programming languages.

CGEN (Birmingham and Siewiorek, 1989), a knowledge-acquisition tool for computer-design systems, and SALT (Marcus, 1988), the knowledge-acquisition tool for the VT system, embody many of the ideas that will be present in DIDS-generated knowledge-acquisition tools. Both CGEN and SALT play an active role during knowledge acquisition by detecting inconsistencies and missing knowledge. We believe that the strong model of mechanisms and knowledge forming the foundation of DIDS will enable DIDS-generated knowledge-acquisition tools to have these features. CGEN and SALT, however, were built to support a particular PSM. Thus, they have limited applicability. DIDS removes these limitations and incorporates VT, mechanisms and PSM in its libraries.

Chandrasekaran proposes a model of design based solely on the concept of *generic tasks* (Chandrasekaran, 1986, 1990). Generic tasks decompose design tasks hierarchically. Each task is defined by its position in the hierarchy, the method used to perform it, and the knowledge, both declarative and control, required to perform the task. Generic tasks have three principal weaknesses, which are shared with DIDS, but to a lesser degree. DIDS' model of configuration design, which is grounded in a careful study of configuration systems, helps to reduce these problems. First, the generic-task model does not establish which tasks are generic. Second, the design system's task may not decompose neatly into a disjoint set of high-level generic tasks even though there may exist a set of lower-level tasks that could implement the system. For example, it is possible that a design system might first perform half of the classification generic task, then do a critiquing task and then finish the classification. Finally, the separate implementation of each generic task requires an environment to integrate and to allow communication between tasks. Also, a system implemented using generic tasks may contain multiple copies of the same piece of knowledge, since the knowledge used by two generic tasks may overlap. The environment must ensure that the knowledge contained in different generic tasks is consistent.

7. Summary

The craftsmanship required for developing knowledge design systems has hampered the widespread use of this technology. For a cogent task, such as configuration design, craftsmanship can be replaced by partial automation. Accordingly, the DIDS model for rapidly creating configuration design systems was developed.

DIDS is based on the idea of reusability of knowledge structures and mechanisms. By utilizing an integrating framework, mechanisms and knowledge structures can be quickly applied to new application domains. Furthermore, these systems can be assembled without detailed programming knowledge. The DIDS approach was demonstrated in this paper on an elevator-design problem, which is relatively large (hundreds of constraints and parts). We have recently used the DIDS approach on a number of other problems, the largest being a computer configuration task, that has over 10 000 constraints and over 1500 components.

Acknowledgements

This work was funded, in part, by a gift from Digital Equipment Corporation and by the National Science Foundation Grant MIPS-905781. The opinions expressed in this paper are those of the authors, and do not necessarily reflect those of Digital Equipment Corporation or NSF. Manjote Haworth was instrumental in developing the constraint network.

References

Balkany, A., Birmingham, W. P. and Tommelein, I. D. (1993) An analysis of several design tools. *Artificial Intelligence in Engineering, Design, and Manufacturing*, **7**(1), 1–17.

Barker, V. and O'Conner, D. (1989) Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, March.

Birmingham, W. P. and Diewiorek, D. (1989) Automated knowledge acquisition for a computer synthesis system.

Birmingham, W. P. and Tommelein, I. D. (1992) Towards a domain-independent synthesis system, in *Knowledge Aided Design*, Green, M. (ed.), Academic Press, London.

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

Boehm, B. W. (1987) Improving software productivity. *Computer*, September, **20**(9), 43–57.

Brown, D. and Chandrasekaran, B. (1987) *Design Problem Solving – Knowledge Structures and Control Strategies*, Morgan Kaufmann, San Mateo, CA.

Brown, D. C. and Chandrasekaran, B. (1989) *Design Problem Solving: Knowledge Structures and Control Strategies*, Morgan Kaufmann, Pitman Publishing, London.

Chandrasekaran, B. (1986) Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *AI Magazine*, Fall.

Chandrasekaran, B. (1990) Design problem solving: a task analysis. *AI Magazine*, Winter.

Dechter, R. and Pearl, J. (1987) Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, **34**(1), 1–38.

Doyle, J. (1979) A truth maintenance system. *Artificial Intelligence*, **12**, 231–272.

Johnson, M. V. Jr and Hayes-Roth, B. (1988) Learning to solve problems by analogy, Report No. KSL-88-01, Stanford University, Department of Computer Science, Knowledge Systems Laboratory.

Haworth, M. S., Birmingham, W. P. and Haworth, D. E. (1992) Optimal part selection, CSE-TR-127-92, University of Michigan, Computer Science and Electrical Engineering Division.

Klinker, G., Bhola, C., Dallemagne, G., Marques, D. and McDermott, J. (1990) Usable and reusable programming constructs, in *Proceedings of the 5th Knowledge Acquisition Workshop*, AAAI.

Langrana, N., Mitchell, T. and Ramachandran, N. (1986) Progress towards a knowledge-based aid for mechanical design, in *Symposium on Integrated and Intelligent Manufacturing*, The American Society of Manufacturing Engineers.

Mackworth, A. K. (1977) Consistency in networks of relations. *Artificial Intelligence*, **8**(1), 99–118.

Maher, M. L. (1987) Engineering design synthesis: a domain independent representation. *AI EDAM*, March.

Marcus, S. (ed.) (1988) *Automating Knowledge Acquisition for Expert Systems*, Kluwer, Boston.

Marcus, S., Stout, J. and McDermott, J. (1987) VT: an expert elevator designer that uses knowledge-based backtracking. *AI Magazine*, Winter.

McDermott, J. (1980) R1: a rule-based configurer of computer systems, No. CMU-CS-80-119, Department of Computer Science, Carnegie Mellon University.

McDermott, J. (1988) Preliminary steps towards a taxonomy of problem-solving methods, in *Automating Knowledge Acquisition for Expert Systems*, Marcus, S. (ed.), Kluwer, Boston.

Mittal, S. and Araya, A. (1987) A knowledge-based framework for design, in *Proceedings of the 5th National Conference on AI*, pp. 856–865.

Mittal, S. and Frayman, F. (1989) Towards a generic model of configuration tasks, in *Proceedings of the 11th IJCAI*, August, pp 1395–1401

Musen, M. (1989) *Automated Generation of Model-Based Knowledge-Acquisition Tools*, Morgan Kaufmann, San Mateo, CA.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T. and Swartout, W. (1991) Enabling technology for knowledge sharing. *AI Magazine*, **12**(3), 36–56.

Neighbors, J. (1984) The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, September.

Newell, A. (1981) The knowledge level. *AI Magazine*, Summer.

Puerta, A. R., Egar, J. W., Tu, S. W. and Musen, M. A. (1992) A multiple method knowledge-acquisition shell for the automatic generation of knowledge-acquisition tools. *Knowledge Acquisition*, **4**(2), 171–196.

Rich, C. and Waters, R. (1988) Programmer's apprentice: research overview. *IEEE Computer*, November.

Rosenblatt, A. and Watson, G. (eds) (1991) Concurrent engineering. *IEEE Spectrum*, July, 22.

Runkel, J. T. and Birmingham, W. P. (1992) Knowledge acquisition in the small, in *Proceedings of the AAAI Knowledge Acquisition for Knowledge-based Systems Workshop*, Banff, October.

Runkel, J. T., Birmingham, W. P., Darr, T. P., Maxim, B. R. and Tommelein, I. D. (1992) Domain independent design system: environment for rapid prototyping of configuration design systems, in *Proceedings of the 2nd International Conference on Artificial Intelligence in Design*, AID 92, 22–25 June, Pittsburgh, PA, Gero, J. S. (eds), Kluwer, Dordrecht, pp. 21–40.

Sussman, G. J. and Steele, G. L. Jr (1980) CONSTRAINTS – a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, **14**, 1–39.

Tommelein, I. D., Levitt, R. E. and Hayes-Roth, B. (1992) SightPlan model for site layout. *Journal of Construction Engineering and Management*, **118**(4), 749–766.

Tong, C. (1987) Towards an engineering science of knowledge-based design. *Artificial Intelligence in Engineering*, **2**(3).