Knowledge Acquisition (1993) **5**, 221–243

Knowledge acquisition in the small: building knowledge-acquisition tools from pieces

JAY T. RUNKEL AND WILLIAM P. BIRMINGHAM

Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109, USA

The knowledge-systems community is interested in easing the knowledge-system development process. One approach, the *mechanisms* approach, views knowledge systems as a set of tasks, each of which can be realized by a computation mechanism. To be effective, knowledge-acquisition (KA) tools must be automatically configured once a set of mechanisms has been selected. We present a method for automatically generating a model-based KA tool for a given set of mechanisms. The method advocates combining *KA mechanisms*, which acquire knowledge in the small, and a set of strategies that provide a global view of the KA activity. We show that these global strategies are necessary for the KA tool to efficiently interact with a domain expert.

1. Introduction

The knowledge-systems community has recently become interested in easing the knowledge-system development process. Several proposals for doing this have been offered, including translation languages (Neches *et al.*, 1991) and what we shall term *mechanisms* (Marques *et al.*, 1992). The mechanism approach, with roots in Generic Tasks (Chandrasekaran, 1986), Steel's (1990) componential framework, and problem-solving methods (PSM) (McDermott, 1988), views knowledge systems as a set of tasks, where each task can be realized by a computation mechanism. It has been postulated that many of these tasks are domain independent, or *reusable*. There has been some empirical evidence to support this postulation; a number of researchers have successfully ported systems from one domain to another by modifying only the domain-specific portions of the knowledge base (Langrana, Mitchell & Ramachandran, 1986; Brown & Chandrasekaran, 1987; Maher, 1987; Johnson & Hayes-Roth, 1988; Birmingham & Tommelein, 1991). Recently, an analytical study has demonstrated that common mechanisms exist among a number of configuration-design systems (Balkany *et al.*, 1993*a*).

The mechanisms approach has several advantages. Once identified and made reusable, mechanisms can significantly reduce programming effort (Klinker *et al.*, 1990; Neches *et al.*, 1991; Runkel *et al.*, 1992). That is, mechanisms can be reused among a variety of systems. Furthermore, if mechanism behavior can be described in the language used by domain experts, then they may be able to construct systems themselves, given appropriate tools (Klinker *et al.*, 1990).

To reap the potential advantages of mechanisms, software-development systems must be constructed to support their integration and form a complete knowledge system. Depending on the intended user—domain expert or knowledge engineer—the interface to such a system will need to be customized. We are concerned with creating a software-development system, called DIDS† (Birmingham & Tommelein,

† Domain-Independent Design System.

1042-8143/93/020221 + 23$08.00/0 © 1993 Academic Press Limited

1991; Runkel & Birmingham, 1992; Balkany *et al.*, 1993*b*), to help knowledge engineers construct systems faster by automating much of the requisite programming. Through an interactive process, a knowledge engineer using DIDS constructs a PSM from a library of mechanisms. This results in a knowledge system and a knowledge-acquisition (KA) tool. The KA tool is then used by a domain expert to develop a complete knowledge base.

Model-based KA tools are particularly well suited for acquiring knowledge in domains where DIDS-generated knowledge systems are used. These tools have demonstrated the ability to create large complex knowledge bases, in some cases without requiring the knowledge engineer's intervention. These tools work by exploiting a model of the problem solver and its knowledge structures. The model describes how to acquire and then structure knowledge (Birmingham & Klinker, 1993). This model, however, has several limitations: it is hand-crafted by a knowledge engineer familiar with the problem solver, it is specific to a particular problem solver and knowledge structure, and it is highly compiled. This creates a fundamental problem: PSMs are built on a *per application* basis, so each DIDS-generated knowledge system may have a different PSM. Any potential gains afforded by rapid programming would be abrogated by requiring a user to construct a detailed model of the PSM in order to build a KA tool.

Thus, a means for developing model-based KA tools without using handcrafted, explicit models must be found. We propose that by combining knowledge of PSMs with properties of the underlying knowledge structure, this problem can be ameliorated. Specifically, KA procedures similar to mechanisms can be revised to rapidly construct KA tools that drive the acquisition process using the knowledge roles of mechanisms, that perform consistency and completeness checks, and that generalize and operationalize knowledge (Birmingham & Klinker, 1993). We associate model-based acquisition procedures, called *mechanisms for KA* (MeKA) (Marques *et al.*, 1992) with (PSM) mechanisms, and general-purpose acquisition procedures with knowledge structures. MeKAs alone, however, cannot produce the capabilities of model-based tools. Global acquisition strategies, ensuring that questions are posed in an order that makes sense to the user, must be applied to sequence the MeKAs.

Like mechanisms, MeKAs have the properties of reusability and combinability. This allows a library of MeKAs to be built and consequently reused among many knowledge-acquisition tools. Thus, the MeKA library reduces the time to create a knowledge-acquisition tool as a mechanism library does for a knowledge system (see Section 5).

We are limiting our initial investigation of mechanisms to configuration-design systems. Roughly speaking, these systems† *construct* a design from a fixed set of parts, which may be stored as a library, or given as input. These parts are well characterized with respect to function and relationships among each other. During configuration, parts are selected, and then interconnected or arranged in a topological or sometimes geometrical order such that the user's specifications are met. Configuration design is an ideal starting point because the definition of the task is cogent, there are a large number of working systems that perform the task, and it is inherently diverse, *i.e.* not all configuration tasks are the same.

† See Mittal and Frayman (1989) for a formal definition.

In this paper, we expand on these issues, and present ideas for developing model-based KA tools from mechanisms. Our framework of knowledge systems is explained in Section 2 and MeKAs are described in Section 3. The use of strategies is presented in Section 4, and our preliminary results are discussed in Section 5. Related work is presented in Section 6, and a summary is given in Section 7.

2. The DIDS model

In this section, we present the DIDS model of knowledge systems; the model strongly influences our view of how KA tools can be assembled. The model is based on three elements: mechanisms, knowledge structures, and process models. Mechanisms are software routines that implement the subtasks of a knowledge system. They communicate with each other using the set of knowledge structures. The process model contains a set of data structures and inference methods on these structures that operationalize the knowledge structures and mechanisms. The data structures and inferences in the process model are geared towards efficient problem solving, unlike the knowledge structures, which are geared to facilitate knowledge acquisition. Each model element is described below.

2.1. KNOWLEDGE STRUCTURES

The DIDS model defines a set of ten knowledge structures that identify what knowledge is required to perform configuration design; other tasks could require different structures. These knowledge structures were identified by studying existing configuration-design systems (Balkany *et al.*, 1993*a*), and we believe that they are sufficient to represent all the domain knowledge necessary for all configuration tasks. More importantly, however, is that the knowledge structures provide guidance for assembling MeKAs in a meaningful order, as described in Section 4.

The knowledge structures define the functionality of mechanisms, the knowledge communicated between mechanisms, and the domain knowledge used by mechanisms. All mechanisms are defined by the operations that they perform on the knowledge structures; the inputs and outputs of mechanisms must be in terms of the knowledge structures. For example, as defined below, there are knowledge structures called abstract parts, parts, and connections. A mechanism may take an abstract part (i.e. a function) as an input and return a part that implements it; alternatively, a mechanism can be given a set of parts and determine the connections among them.

The knowledge structures assumed by DIDS are the following:

1. **Parts:** The part knowledge structure represents the elements in the part library. Parts are defined by a set of attributes and ports. The attributes of a part define the properties of a part that can be expressed by a name and a scalar value. Ports define how a part can be connected to other parts. The attributes, which are called *characteristics*, have values that are defined before problem solving begins and cannot change during problem solving. In addition, each part must realize at least one function, so that its addition moves the design closer to completion.
2. **Function:** Functions define what is required of the artefact being designed for a particular problem instance. This knowledge structure is needed for the

part-selection subtask of design. It drives the design process, as parts are selected to provide the functions required.

3. **Abstract Parts:** Abstract parts represent all the functions and subfunctions that an artefact being designed may perform. Abstract parts are defined by their characteristics, ports, and specifications. Specifications are attributes whose values depend on the problem instance, and, therefore, their values must be computed during problem solving. This knowledge structure provides a hierarchical decomposition of the functions required of the artefact.
4. **Subfunction:** The subfunction knowledge structure successively decomposes the artefact being designed along functional lines. It describes the functional relationship between the parts and abstract parts. This relationship describes how abstract parts may be realized by combining sets of abstracts parts and parts.
5. **Required Functions:** Parts and abstract parts often require functions performed by other parts to support their operation. This information is contained in the required-function knowledge structure. Associated with each function performed by a part is a list of required functions that must be realized by the artefact. The part will not realize its intended functions unless the artefact realizes the required functions. Required functions do not add any desired functionality to the artefact; rather, they perform a function that is necessary for other parts to operate.
6. **Constraints:** Constraints specify algebraic relationships among 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 attributes' values.
7. **Preferences:** 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 produced.
8. **Ordering:** Problem solvers use task-ordering knowledge to determine the most effective order of designing the abstract parts. For some problems, the order in which subtasks are performed affects both the quality of the design and the amount of search needed to generate it.
9. **Connections:** Connection knowledge constrains the set of possible connections that can be made among the ports of parts and abstract parts. It may either specify illegal connections, or sets of connections that have been found to be useful in the past.
10. **Arrangement:** Arrangement knowledge specifies how parts can be geometrically or topologically arranged. It constrains the positions a part may occupy.

The knowledge structures are combined to form a graph, which is the knowledge base. Two structures form nodes in this graph: part and abstract part. The other structures represent relationships between them. For example, Figure 1 shows a knowledge base for a bicycle task. This figure contains both parts and abstract parts and two sets of links. First, the subfunction knowledge structure defines the functional relationships among the parts; in this view, the parts form a hierarchy.

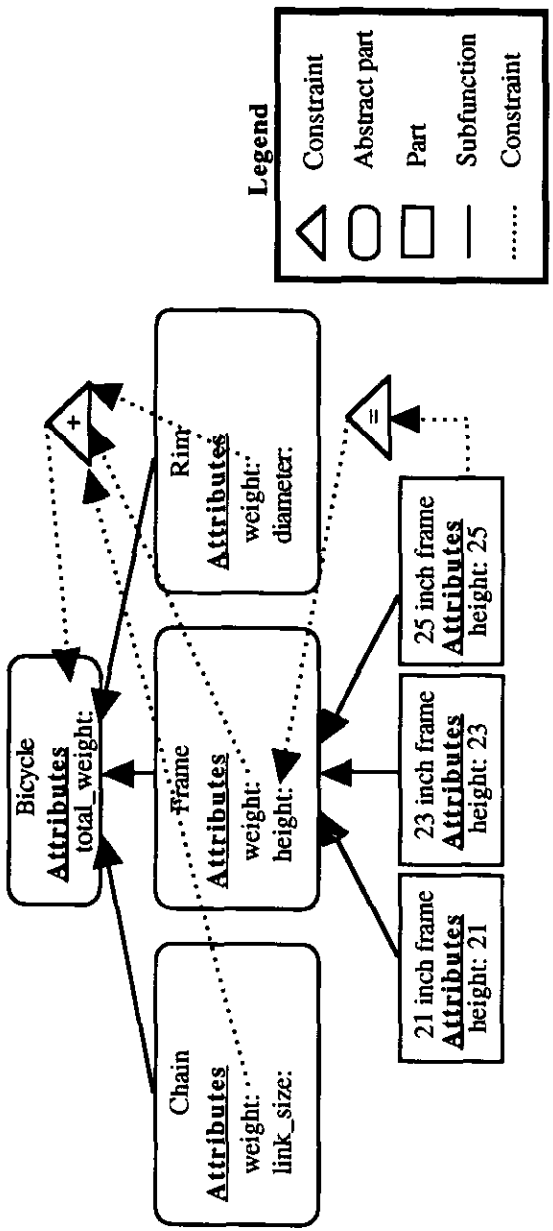

FIGURE 1. A portion of a bicycle knowledge base.

Second, attribute constraints relate the capacities of various components. The figure contains two constraints: one that computes the total weight of the bicycle by summing the weights of the chain, frame, and rim, and another that sets the weight of the frame equal to the weight of the part chosen to implement the frame.

2.2. MECHANISMS

Underlying our work is the notion that a configuration system can be represented as a collection of mechanisms. Many of these mechanisms can be separated from task-specific knowledge, and therefore made reusable. Mechanisms are made operational for a new task by *adding* task-specific knowledge to them.

As described above, mechanisms are operators on the knowledge structures. This has two advantages. First, the mechanisms will be reusable and combinable. This definition ensures reusability, because it places no restrictions on the task-specific concepts that must be supplied, or on the source task of the concepts. The only requirement is that the knowledge-structure classification of the concepts is the same as the inputs to the mechanism. This guarantees that a mechanism can be applied to any configuration task where the task contains the appropriate knowledge structures. The knowledge structure definition also ensures the combinability of mechanisms, since all the mechanisms share a common representation of these structures. Therefore, any two mechanisms that use the same knowledge structures can share information, and can be easily combined. Second, this definition makes clear exactly which 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 PSM and to guide the construction of a knowledge-acquisition tool for the method.

Mechanisms implement techniques for solving design problems. Two mechanisms may perform the same function (e.g. part selection), but may differ in the algorithm or knowledge structures they use. Each mechanism is implemented by a code fragment, and is associated with a procedure for acquiring the task-specific knowledge required for that mechanism to operate, i.e. a MeKA (discussed more fully in Section 3). Each mechanism is characterized by a set of task features that describe when it should be used, and a description of its inputs and outputs. For example, Figure 2 shows the pseudo code and input and output parameters for a mechanism that performs part selection. Mechanism parameters are represented in the form: (⟨parameter-type parameter-name⟩ . . .).

To be useful, mechanisms must be combined within a control structure, thereby forming a PSM. For example, Figure 3 shows a PSM generated by DIDS to perform a bicycle-design task. The mechanisms are highlighted in boldface, and the

```
select_candidate_parts(<Abstract-Part nd>, <Abstract-Parts candidates>)
Get nd's children in the functional hierarchy;
For each of child, ch, of nd
        if ch satisfies the constraints on nd then
                add ch to candidates;
```

FIGURE 2. An example mechanism to select parts from a hierarchy.

```
Abstract-Parts undesign-functions;          //Queue of functions, called abstract parts,
                                            //that need to be designed.
DesignState ds;                             //Data structure containing the artefact.
Abstract-Part current-function,             //The current function being designed in this
                                            //iteration of the PSM.
              newpart;                      //The part or abstract parts selected as the design of
                                            //the current function.
Abstract-Parts candidate-subfunctions;      //Set of parts or abstract parts that implement the
                                            //current function and satisfy all the constraints.

load_kb("bicycle.dids.kb");                 //Loads the KB created by the KA tool.
initialize_queue(undesigned-functions);     //Puts the first abstract-part "bicycle" into the
while (not_empty_queue(undesigned-functions))  //queue.
{
   apply_to_queue(compute_spec_values, undesigned-functions);   //Applies compute_spec_values
                                                                //mechanism to each function in the
                                                                //queue
   get_next_function(undesigned-functions, current-function);   //Gets the first function in the queue.
   add_to_design(ds, current-function);                         //Adds the function to the design.
   if (is_an_AND_NODE(current-function))
   {
      select_all_parts(current-function, candidate-functions);  //Selects all the current-function's
                                                                //subfunctions
                                                                //and adds them to the queue.
      add_to_queue(undesigned-functions, candidate-functions);  //selects the subset of the current-
   }                                                            //function's children that satisfy the
   else if (is_an_OR_NODE(current-function))                    //constraints
   {
      select_candidate_parts(current-function, candidate-functions);
      select_best_part(candidate-functions, newpart);   //Selects the part with the least cost.
      add_to_queue(undesigned-functions, newpart);      //Adds the part to the queue.
   };
}
```

FIGURE 3. The bicycle configuration PSM.

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 of the outer WHILE loop.

The mechanisms in this PSM maintain a queue of functions (the abstract-part knowledge structure is used to represent these functions) that are needed in the bicycle, but that have not been completely designed. The PSM assumes that the library of parts in its knowledge base has been organized into a functional hierarchy consisting of *or* and *and* nodes. The PSM begins each design cycle by computing the values of the attributes of all the abstract parts in this queue. The *compute_spec_values* mechanism, which uses constraints associated with attributes, calculates the attributes' values. During each design cycle, the PSM designs the first abstract part in the queue, called the *current-function*. If the current-function is an *and* node, then the *select_all_parts* mechanism adds all of the current-function's children to the queue. If the current-function is an *or* node, then the *select_candidate_parts* mechanism determines the subset of the current-function's children that satisfy the constraints. Then, the *select_best_part* mechanism selects the child that has the lowest cost. Finally, this abstract part is added to the queue.

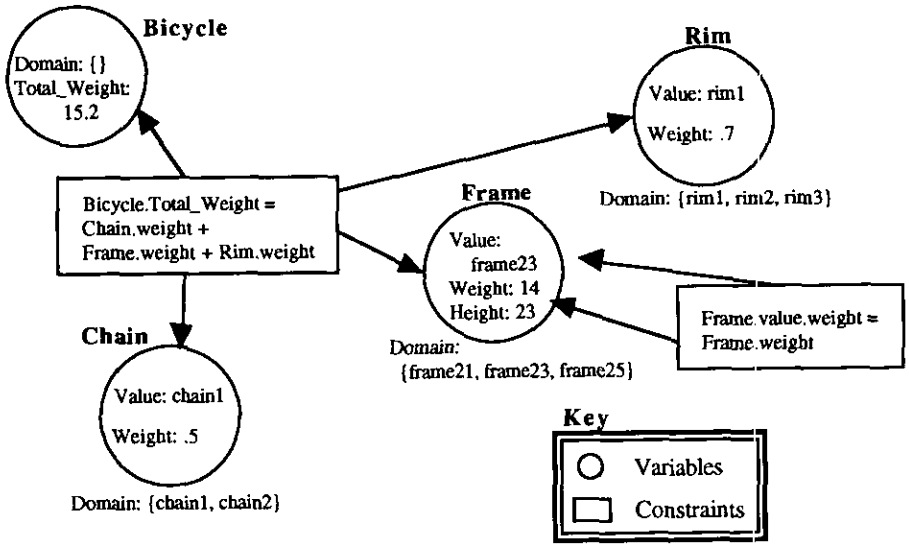

FIGURE 4. Constraint network model representation of a knowledge base.

2.3. THE PROCESS MODEL

The process model provides both a mapping from knowledge structures to data structures and basic inference techniques to support a PSM. The model provides a set of data structures that can be used to represent each knowledge structure. The inference techniques support mechanisms, but are not usefully represented as mechanisms because they do not require task-specific knowledge for their operation, and they rely on a particular knowledge representation that is not portable.

Currently, DIDS has two alternative process models: tables (Haworth *et al.*, 1992) and a constant network. Tables are best used for problems where parts are easily organized as a hierarchy, and finding parts to cover high-level functions is the primary consideration. The constraint network is applicable to problems where problem solving is dominated by constraint satisfaction. The table and network representations of the knowledge base in Figure 1 are shown in Figures 4 and Table 1. In the constraint-network process model (Figure 4), abstract parts such as frame, chain, and rim become variables in the network. The domains of these variables are

TABLE 1
Table-based model representation of a knowledge base

	Chain	Frame	Rim
Chain 1	1		
Chain 2	1	1	
Frame 21		1	
Frame 23		1	
Frame 25			
Rim 1			1
Rim 2			1

the parts that can be used to implement them, i.e. the parts below the abstract part in the knowledge-structure graph. The constraints between abstract parts in the knowledge structure are transformed into constraints between variables in the process model, and are used to restrict the assignment of parts to variables.

In the table process model (Table 1), the parts form the rows of the table and the abstract parts become the columns of the table. Numbers in the cells, which correspond to the solid lines with arrow heads in Figure 1, indicate the number of functions that are implemented by each part. The constraints, which are not shown in the figure, are applied after a set of parts that covers all the necessary functions has been selected.

Each process model has supporting inference methods associated with it. For example, the constraint-network process model has node-consistency and arc-consistency (Mackworth, 1977) inference methods. As node- and arc-consistency operations are tightly linked to a particular model, they are not especially reusable for other process models. Hence, inference methods of this type are not cast as mechanisms.

3. Mechanisms for knowledge acquisition

In the next two sections, we describe how a model-based knowledge-acquisition tool is constructed by using the MeKA library. This section motivates and describes MeKAs, and presents an example of their use. This example demonstrates the need for a construct to control the sequencing of these MeKAs. The next section presents this sequencing construct, called a *knowledge-acquisition method* (KAM).

3.1. MODEL-BASED MeKAs

Model-based KA tools provide powerful support by exploiting the knowledge structures used by PSMs and the roles that these structures play during problem solving. This enables the tools to acquire knowledge in a order that facilitates effective acquisition, to find incompleteness and inconsistency errors, to generalize knowledge, and to acquire knowledge using task specific interfaces (Birmingham & Klinker, 1993). These capabilities are not found in either more general-purpose tools or task-specific tools that do not assume a particular method.

In order to exhibit these same behaviors, each MeKA uses both the knowledge structures and roles assumed by its associated mechanism to drive the acquisition process, and a five-step process for acquiring knowledge. Each of the steps corresponds roughly to one of the tasks performed by model-based KA tools. This five step process is shown below:

1. If possible, infer new knowledge from previously acquired knowledge.
2. Present information that the user is likely to want to review before entering the new knowledge.
3. Ask user for new information, and acquire it.
4. Verify that the knowledge provided by the user meets the mechanism's requirement.
5. If possible, generalize this knowledge to populate other areas of the knowledge base.

A MeKA begins the acquisition process by determining if it can use already acquired knowledge to infer new knowledge. If so, the MeKA performs the inference. Otherwise, the MeKA must acquire the knowledge from the domain expert. The MeKA presents the portion of the knowledge base that the domain expert would likely want to review during acquisition, and then asks the domain expert for the new knowledge. Once knowledge has been acquired, the MeKA verifies the knowledge, ensuring that it is both consistent and complete, and, if possible, generalizes this knowledge to cover other areas of the knowledge base.

A MeKA has five components—infer, present, acquire, verify, and generalize—that correspond to the five steps above. The *infer* component uses a MeKA-specific inference procedure to automatically derive the necessary knowledge. The *present* component of a MeKA presents the relevant sections 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 him with the complexities of the knowledge base. The *acquire* component either asks the user to modify the section of the knowledge base displayed by the acquire component, or asks the user for some additional information about this section. This component uses the knowledge roles of its mechanism to address the expert with a meaningful question.

The present and acquire components use a variety of interface styles to acquire knowledge from a user. These interfaces include:

1. *interactive dialog*: A text-based strategy where the user is asked a question, and responds by selecting from a menu or typing a response.
2. *graphical*: A graphical presentation strategy where the knowledge base is presented to the expert who modifies or extends it.
3. *table*: A cross between interactive and graphical strategies where a partially complete table is presented to the domain expert who fills in the missing elements.

For example, Figure 5 shows an interface that uses a combination of interactive dialog and graphical styles, and examples of table-based styles are given in Section 3.3.

After the knowledge has been acquired, the *verify* component performs the model-based consistency and completeness checks described above and the *generalize* component attempts to generalize this knowledge. If a problem is found in the verification step, the user has the option of either revisiting the present and acquire components, or postponing the MeKA until a later time.

Every MeKA has an argument, called the focus, that is an instance of knowledge structure determining what concept the MeKA will inquire about. This is the object that is operated on by the MeKA's five components. Figure 5 shows the interface presented by the select-candidate-parts MeKA when acquiring knowledge about the frame abstract part. In this example, frame is the focus of the MeKA.

The select-candidate-parts MeKA (Figure 6) acquires for an abstract part, its focus (D), the set of parts that may be used as the design of this abstract part (A, B and C) and a constraint used to select one of these parts. This MeKA does not have an infer component, so all knowledge must be acquired from the user. The present component filters out a large portion of the knowledge base, and just displays the

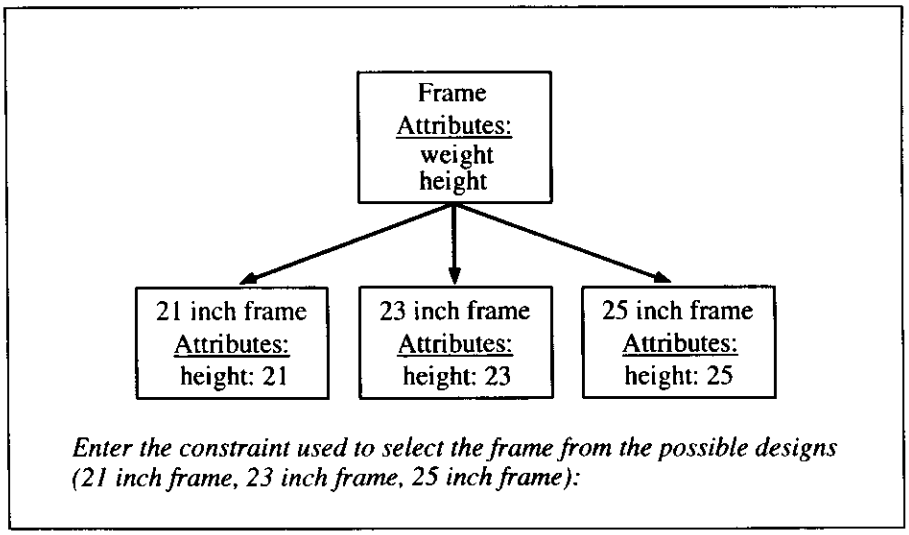

FIGURE 5. Interface generated by a select-candidate-parts MeKA.

focus and the parts related to the focus by the subfunction relation. The diagram shown next to the present component in Figure 6 depicts an example of how these knowledge structures are displayed to the user. For example, if the knowledge base is as shown in Figure 1 and frame is the focus, then the present component would display a two-level hierarchy with frame at the root and with leaves consisting of 21-inch-frame, 23-inch-frame, and 25-inch-frame (Figure 5). The acquire component allows the user to modify the hierarchy displayed by the present component, and asks the user to supply a constraint that can be used to select a frame. An example of a selection constraint would state that the type of the bicycle must match the type

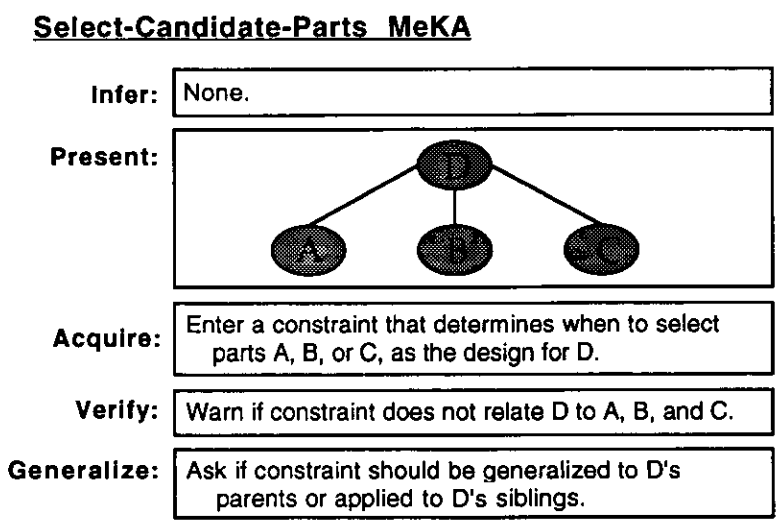

FIGURE 6. The MeKA for the select-part mechanism.

of the selected frame. The verify component checks to ensure that the acquired constraint can be used to select one of the parts, i.e. it defines a relationship between the focus and the parts. Finally, the generalize component asks the domain expert if this constraint can be used elsewhere in the knowledge base. For example, if the domain expert had entered a constraint specifying that the least expensive part should be selected, then he may want to use this constraint elsewhere.

3.2. NON-MECHANISM-SPECIFIC MeKAs

Not all MeKAs use model-based procedures to acquire knowledge for specific mechanisms. A class of MeKAs, called non-mechanism-specific MeKAs, which are present in every KA tool, allow a user to describe the global structure of the knowledge base. The goal of these MeKAs is to gather a high-level picture of the knowledge base that shows its organization and that lists all the domain concepts that must be acquired. The global structure includes the list of all the parts and abstract parts, as well as the subfunction and required-function relations. These MeKAs do not acquire any detailed information about the parts or abstract parts, such as their attributes and constraints among them, just their names. Figure 7 shows an example of a functional hierarchy created by the non-mechanism-specific MeKAs. The interior nodes of this hierarchy represent the abstract parts used to represent a bicycle, and the leaf nodes (not shown in the figure) represent the bicycle parts.

The global structure is not acquired by the model-based MeKAs because they have a local perspective; they describe how to acquire the knowledge for just one mechanism. Since mechanisms operate on a small section of the knowledge base at one time, their MeKAs do not help the domain expert organize the knowledge base. The model-based MeKAs, however, acquire the additional knowledge structures describing the concepts, such as attributes, constraints, and preferences, that are necessary for the mechanisms to operate.

3.3. AN EXAMPLE

The following example, where a bicycle-design system is to be generated (Runkel *et al.*, 1992), demonstrates acquisition power of MeKAs and the need for an intelligent strategy for sequencing them. In this scenario, a domain expert begins by describing the bicycle-design task. Through an interactive dialog, DIDS automatically gener-

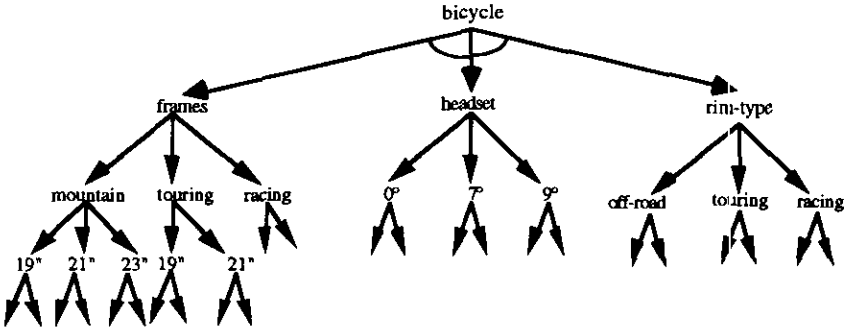

FIGURE 7. Functional hierarchy for bicycle scenario.

ates both the PSM and the KA tool (Balkany *et al.*, 1993*b*). The domain expert uses the KA tool to construct a small, representative knowledge base so that the PSM's performance can be evaluated. During knowledge acquisition, the MeKA's verification components may find situations where acquired knowledge does not match the assumptions made by the PSM, and other shortcomings are found by running test cases. If problems are found, they typically require modifications to the PSM, causing a new KA tool and PSM to be generated. This process continues until the PSM performs to the designer's expectations.

The PSM used in this scenario, shown in Figure 3, requires the following task-specific knowledge: a part library organized into a functional hierarchy, constraints on the attributes of each abstract part and part that can be used to either generate information needed for further design activity (arc-consistency) or to select among (abstract) parts. The KA tool built to acquire this knowledge is shown in Figure 8. It was generated from the MeKAs associated with the mechanisms in the PSM and begins by invoking the non-mechanism-specific MeKAs. (The MeKAs in this KA tool are shown in Figure 9.) Using these MeKAs, the user describes the structure of a bicycle and the parts that compose it by building a functional hierarchy like the one shown in Figure 7.

Once the functional hierarchy has been built, the KA tool orders the model-based MeKAs in the same order as the mechanisms in the PSM. The first node designed by the PSM is the bicycle node, so it becomes the focus for the first sequence of MeKAs. The first mechanism executed in every design cycle is *compute_spec_values*. Therefore, the *compute_spec_values_MeKA* interviews the expert first. The MeKA displays the root node in the hierarchy, in this case the bicycle node, using a table format, and prompts the domain expert for the specification attributes of a bicycle. The domain expert adds a new row to the table for each attribute of a bicycle. (Table 2 shows the completed table.) The verify component of this MeKA requires that each attribute have either a value or a constraint to compute its value. In this case, the *constraints* are questions put to the user to get the design parameters.

```
get_parts_non-mechanism_specific_MeKA
construct_hierarchy_non-mechanism_specific_MeKA
initialize_queue_MEKA
while (more parts in hierarchy to process)      // examine each node in the hierarchy
{
    apply_to_queue_MEKA;                        //this and the next two MeKAs will cause redundant questioning
    get_next_function_MEKA;
    add_to_design_MEKA;
    compute_spec_values_MEKA;
    if Node is AND
        {
        select_all_parts_MEKA;                      // unnecessary MeKA
        add_to_queue_MEKA;                          // unnecessary MeKA
        }
    else
        {
        select_candidate_parts_MEKA;
        select_best_part_MEKA;
        add_to_queue_MEKA;
        }
}
```

FIGURE 8. The bicycle configuration KA tool.

Mechanism	MeKA

compute_spec_values(Abstract-Part part)
{
For each attribute, att, of part
if att does not have a value then
compute att's value using its constraint
}
 	
Task-Selection Features:
Abstract-parts must have attributes related by constraints.
	compute_spec_values-MeKA(*focus*)
	Knowledge used:
	Attributes
	Constraints
	Inference: none
	Acquire:
	Prompt("Use the following table to define the attributes of <*name of focus*>. For each attribute define either a value or a formula that can be used to compute the value of the attribute.")
	Verify:
	{
	for all attributes, att, of focus
	ensure that att either has a value or att has a formula
	}

select_all_parts(Abstract-Part nd,
Abstract-Parts candidates)
{
Get all of nd's children in the functional hierarchy and return them in the candidates set.
}
 	
Task-Selection Features:
The hierarchy must contain and nodes
	select_all_parts_MeKA(*focus*)
	{
	} /* *no knowledge required* */

select_candidate_parts(Abstract-Part nd,
Abstract-Parts candidates)
{
Get *nd's* children, in the functional hierarchy;
For each of child, *ch*, of nd
if ch satisfies the constraints nd then
add *ch* to candidates sets;
Return the candidates set.
}
 	
Task-Selection Features:
The hierarchy must contain or nodes.
	select_candidate_parts_MeKA(Abstract-Part *focus*)
	Knowledge used: Constraints
	Present:
	present_attributes(focus);
	present_attributes(children(focus));
	Acquire:
	prompt("Enter the constraint used to select from the possible designs <*name of focus*>: <*children of focus*>");
	Verify:
	{
	The constraint acquired constrains the focus and its children.
	}

select_best_part(Abstract-Parts candp_arts,
Abstract-Part best)
{
Set best equal to the abstract part or part in the set, cand_parts, that has the cost attribute with the smallest value.
}
 	
Task-Selection Features:
All parts and abstract parts must have cost attribute.
	select_best_part_MeKA(Node *focus*)
	Knowledge acquired:
	Attribute: "Cost"
	Inferences: none
	Acquire:
	prompt("Enter the cost in dollars or a constraint for computing the cost in dollars of the part <*name of focus*>",
	Verify: cost is greater than 0;

FIGURE 9. A few of the MeKAs in the bicycle design system.

TABLE 2
Table for acquiring attributes of bicycle

Attribute	Value type	Units	Value range	Formula
type	STRING	none	mountain, touring, racing	"Enter the type of bicycle you want to design:"
person_height	INTEGER	inches	48 . . . 84	"Enter your height:"
performance	STRING	none	low, moderate, high	"Enter the type of bicycle performance you desire:"

TABLE 3
Table used to acquire the attributes of a frame

Attribute	Value type	Units	Value range	Formula
size	INTEGER	inches	19 . . . 27	bicycle.person_height/4
type	STRING	none	mountain, touring racing	bicycle.type

The next step during problem solving is to determine whether the focus is an *or* or an *and* node. Since *bicycle* is an *and* node, the *select_all_parts_MeKA* is invoked. This MeKA does not acquire any knowledge, so the KA tool proceeds to the children of the bicycle node.

The next several steps of the KA tool operate on the frame abstract part. *Compute_spec_values_MeKa* asks the domain expert to define the attributes of a frame. Table 3 depicts the attributes acquired. At this point, however, the performance of the KA tool degrades due to the simplicity of MeKA ordering. Since frame is an *or* node, the next MeKA is *select_candidate_parts_MeKA*. It acquires a constraint used to select the children of its focus. This constraint compares the attributes of the focus to the attributes of its children, and determines which children can be used in the design. Problems arise because the attributes of the children of frame have not yet been acquired. Therefore, the expert will be unable to give the needed constraint; the KA process can no longer continue. Figure 10 shows the display presented to the user by the *select_candidate_parts_MeKA*.

By rearranging the MeKAs, however, the acquisition process will not have this problem. A better way to construct KA tools is discussed next.

4. Exploiting properties of the knowledge structure

As shown in the previous section, simply integrating a set of MeKAs does not produce a tool that coherently interacts with an expert. Thus, we need to establish a

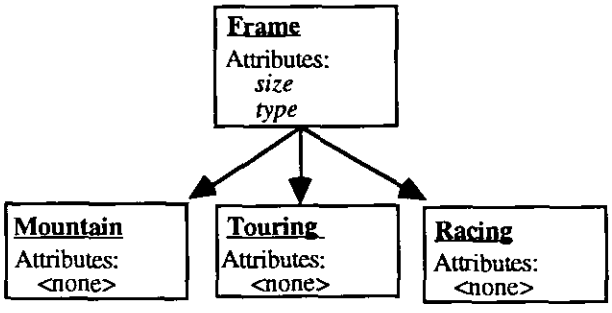

FIGURE 10. Interface presented by select-candidate-parts MeKA to acquire a constraint.

set of acquisition strategies that provide the proper sequencing information. These strategies are derived from the knowledge structures. In this section, we propose a set of strategies.

4.1. INHERENT PROPERTIES OF THE KNOWLEDGE STRUCTURE

The knowledge structures assumed by DIDS are highly related, and these relationships can be exploited for knowledge acquisition. As shown in Section 3, some knowledge structures define relationships among other structures, and, therefore, it is necessary to acquire one knowledge structure before another. In addition, the preferences of domain experts must also be taken into account when acquiring knowledge. A reasonable order for acquiring the configuration-knowledge structures is listed below. DIDS uses this ordering information to determine the best order for invoking the MeKAs in a KA tool. The ordering is called a knowledge-acquisition method (KAM).

A strategy for sequencing MeKAs is given below:

1. *Acquire parts.* Parts form the foundation of any design system, and are easily acquired from catalogs. Parts also determine the types of attributes needed. There are a variety of methods to acquire parts, including user query by forms and optical scanning.
2. *Acquire functions, subfunctions and abstract parts.* Depending on the type of problem solving used, a functional hierarchy may be required. This hierarchy is formed out of *abstract* parts. These abstract parts must be specified by the user, which can only be done once the physical part library has been acquired. The abstraction process used to define abstract parts will, by definition, use subfunction relationships, resulting in a hierarchy. The non-mechanism-specific MeKAs are designed specifically for this purpose. These MeKAs support both top-down and bottom-up hierarchy specification styles, as designers have different preferences.
3. *Acquire required functions, arrangement, and connection knowledge.* These knowledge structures usually go together, and thus should simultaneously be acquired. In addition, they can help to define sections of a hierarchy, so they are often acquired simultaneously with abstract parts. Again, the non-mechanism-specific MeKAs are useful here, as this knowledge is normally graphically specified.
4. *Acquire constraints and preferences.* These are always defined relative to (abstract) part attributes, so the parts must be given first. It is possible, however, to intersperse the acquisition of parts, abstract parts, and constraints and preferences. That is, some parts may be given, then some constraints, and then some more (abstract) parts, and so forth. Under some circumstances, interspersion may be preferable, as it could assist verification. This knowledge can be acquired through a dialog with the domain expert.
5. *Acquire ordering knowledge.* Most often, tasks are associated with parts [e.g. design part X with specification $(s_1, s_2, \ldots, s_n)$]. The ordering knowledge usually acquired here is some specific partial ordering on tasks themselves (design X before Y), or a partial-ordering heuristic (e.g. design big things before small things). In either case, the structure of the hierarchy should be

known before task knowledge is acquired. Note: this knowledge can be acquired anytime after the hierarchy is established.

6. *If a MeKA is used to acquire control knowledge, remove it from any loop.* Many MeKAs will need to acquire knowledge solely for controlling the PSM. In general, this knowledge requires only a simple heuristic, and is not dependent on the particular state of the design, or a part in the library. Thus, it can be removed from any loops in the KAM.

4.2. KAMs

The KAM plays the same role that the PSM plays in the problem solver; it determines how to sequence the MeKAs. The purpose of this sequence, however, is to facilitate acquisition by producing meaningful dialog with the domain expert to extract task-specific knowledge for the knowledge base. In order to fulfill this purpose, a KAM has two responsibilities: it must determine the sequence for invoking the MeKAs and it must select the focus of each MeKA when it is invoked. The MeKA sequence is determined using a set of heuristics, described above, that exploits the dependencies in the knowledge structures. The focus for each MeKA is also determined using a set of heuristics.

A KAM determines the focus of the acquisition process by using one of four strategies. These strategies are tried successively until one is successful. The four strategies are listed below:

1. *Current context.* In order to prevent the KA tool from jumping randomly from topic to topic, the focus of acquisition is kept as consistent as possible. Whenever possible, the focus of the previous MeKA is used as the focus of the next MeKA.
2. *Inconsistent knowledge structure.* Inconsistencies in a knowledge base hinder a user when describing concepts related to the inconsistent concepts and hinder the reasoning processes of the KA tool. Therefore, it is desirable to remove these inconsistencies as soon as possible by making the focus of the next MeKA an inconsistent knowledge structure.
3. *Incomplete knowledge structure.* Incomplete knowledge structures occur when the user has failed to provide all the information about the structure required by the PSM. This prevents the PSM from performing some of the necessary reasoning steps involving the concept.
4. *Present overview, then refine.* Unlike the other strategies, this one encourages the user to select the portion of the knowledge base to define. It presents a high-level picture of the knowledge base, and then allows the domain expert to successively zoom in on a smaller area. The knowledge structure selected becomes the focus of the acquisition process.

4.3. AN EXAMPLE

The KAM in Figure 8 is reorganized into Figure 11 according to the strategies outlined in the previous section. The non-mechanism-specific MeKAs are used to acquire parts and hierarchy knowledge. The other MeKAs are also reorganized. Those performing program control are moved to the end of the acquisition session, and the information they require is asked only once. The first three MeKAs in

```
get_parts_non-mechanism_specific_MeKA
    for each part:                          // Acquiring attributes here eliminates problems described in Section 3.
        compute_spec_values_MeKA    // uses inheritance to reduce questioning
construct_hierarchy_non-mechanism_specific_MeKA
while (more parts in hierarchy to process)      // examine each node in the hierarchy
{
    if Node is OR
        {
        select_candidate_parts_MEKA;
        select_best_part_MEKA;
        }
}
initialize_queue_MEKA
apply_to_queue_MEKA;
get_next_function_MEKA;
add_to_design_MEKA;
add_to_queue_MEKA;
```

FIGURE 11. The bicycle configuration KAM.

Figure 8 have this property, and can be safely removed from the part-inspection loop. In addition, the *add_to_queue_MeKA*, which is visited for each node, has been moved to the end of the KAM.

Finally, the inner loop remains virtually the same, except as noted in the preceding paragraph. The constraint information for each part is still acquired in the loop. Since all the attributes have been acquired, however, the problems noted in Section 3 will not occur.

The result is a KA process that flows much more smoothly. The same type of knowledge is acquired at the same time, and in the correct order. Furthermore, redundant questions are eliminated.

5. Preliminary results

An important aspect of the DIDS project, and particularly the work discussed here, is reusability. It requires significant, albeit a reasonable amount, of work to develop a MeKA library. Thus, building this library makes sense only if it can be used for many applications. We have applied DIDS to three distinct tasks, whose data was supplied by parties outside our research group, to get a sense of the system's reusability and capabilities. We have not finished implementing systems for all these tasks yet, but some initial results are available. Before discussing these results, the tasks are described below:

- *Elevator task*: this task involves constructing an elevator system from a part library that is consistent with a set of constraints and preferences (Yost, 1991, pers. comm.). The task involves hundreds of constraints and parts. We have fully implemented this system.
- *Room assignment*: for this task, a group of people are assigned offices consistent with constraints and preferences (Linster, 1992, pers. comm.). The task is relatively small, consisting of several constraints and about a dozen people, but requires extensive search. This task has been fully implemented (Balkany *et al.*, 1993*c*).

- *Personal-computer configuration*: in this task, the user presents to the system a set of parts (disk drives, processors, memory chips, etc.) that he believes forms a working personal computer. The system must verify that these parts can, in fact, be combined to form a working PC using a set of constraints describing the legal connections among parts. The task is large, involving thousands of constraints and parts. We have partially implemented this system.

The knowledge structures have adequately represented all the knowledge needed to perform each task. This is a significant finding, as the task-specific knowledge for each task is very different. Furthermore, the mechanisms used to solve these tasks were reused. It is interesting to note that the PSMs for the elevator task and the room-assignment task are very similar, differing in only a single mechanism. Similarly, the KAMs for both tasks are nearly identical. The same is true (so far) for the PC tasks, although both the PSMs and KAMs are significantly different from the first two tasks.

Table 4 compares the size of the three tasks, the amount of reuse, and the development time required to automate them. The room-assignment task was implemented first, when the mechanism library did not contain any mechanisms, so there was not any reuse for this task and a large amount of time was spent building its PSM. The elevator task reused all six of these mechanisms, but required an additional one. Significant time was spent building the knowledge base for this task because of the large number of complex constraints. Finally, the PC task only required several of the mechanisms from the library and no new ones. The knowledge base, however, was supplied by an industrial affiliate and the DIDS-generated KA tool was used only for browsing.

Although the results are preliminary, two claims can be made. First, it appears to be feasible to reuse mechanisms and MeKAs and this reuse saves development time. Second, the large size of the VT and PC tasks has demonstrated that this approach scales to large real-world knowledge systems.

TABLE 4
Reusability of mechanisms and MeKAs†

	Task size		Reuse		Development time	
Task	Number of parts	Number of constraints	Number of mechs. used	Number of mechs. from library	Man-hours to build PSM	Man-hours for knowledge acquisition
Room Assign.	30	4000	6	0	10	2
Elevator	200	500	7	6	1	250
PC	1000	1000	3	3	1	—

† Numbers in this table are estimates.

6. Discussion

DIDS shares many features with systems that automate the development of general software. DIDS falls between domain-specific program generators that automatically create applications in a narrow domain, and systems that use massive code or program schema libraries, which cover broad domains, to assist programmers (Krueger, 1989). DIDS, like application generators, automatically generates source code. DIDS, however, does not have the severe narrow-domain restrictions that limit the usefulness of most application generators (Horowitz, Kemper & Narasimhan, 1985). DIDS achieves this additional functionality by providing a library of basic elements (knowledge structures, mechanisms, and MeKAs) that represent the building blocks of a class of systems. The flexibility of the elements enables a broad range of systems to be generated, and the well-defined uses of these elements allows the generation process to be automated. Systems that reuse code or program schemas resort to a semi-automated development process due to the large number of elements in the library and small number of constraints restricting the selection and combination of these elements (Rich & Waters, 1990).

Klinker *et al.* (1990) have created a system called Spark, Burn and FireFighter (SBF), which is based on combining mechanisms, similarly to DIDS. The two approaches differ in that SBF 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 knowledge structures and the set of mechanisms required to construct systems. Instead, SBF assists a developer with creating and updating a vocabulary that is shared among task activities and mechanisms. The vocabulary is used as an indexing scheme into a library of mechanisms; a more detailed description can be found elsewhere in this issue. Burn, the component of SBF, construct tools by combining computational KA mechanisms, which are analogous to DIDS's MeKAs. These mechanisms are ordered using the sequence of mechanisms in the PSM, similar to the method discussed in Section 3.

PROTÉGÉ II (Puerta, Tu & Musen, 1992; Eriksson *et al.*, 1992) also utilizes a mechanism-like model. It contains a library of both tasks and mechanisms. Links, which connect tasks to mechanisms, suggest mechanisms capable of performing a user's task. PROTÉGÉ II associates with each mechanism an arbitrarily complex data model, instead of a standard set of knowledge structures like DIDS. Each mechanism has a set of editors, similar to MeKAs, capable of acquiring its model. Once the mechanisms have been selected, PROTÉGÉ II generates a KA tool by combining the editors associated with the mechanisms. The emphasis on building KA tools that are model-based is not part of PROTÉGÉ II.

KADS (Wielinga, Schreiber & Breuker, 1992) establishes a methodology for constructing knowledge-based systems based upon a model of expertise similar to that of DIDS, but the components of the KADS model are not operational. The KADS model contains domain knowledge, inference and task knowledge, and strategic knowledge, which are concepts that are analogous to knowledge structures, mechanisms, and PSMs, respectively. Like DIDS, knowledge systems are constructed by reusing computational constructs, and then encoding the domain knowledge in a form usable by these constructs. The level of automation and its scope distinguish KADS from DIDS. KADS is primarily a methodology, with some

tool support. DIDS, on the other hand, provides a greater level of automation of the process of building knowledge systems, but for a smaller range of tasks.

DIDS automates the process of building model-based KA tools for design tasks, such as CGEN (Birmingham & Siewiorek, 1989; Birmingham, Gupta & Siewiorek, 1992) and SALT (Marcus, 1988). Like these tools, DIDS uses a model of both the PSM and knowledge assumed by the PSM to derive its power. The knowledge structures define the possible knowledge roles that must be acquired by the KA tool. MeKAs associated with each mechanism determine which of these knowledge roles are required, how to best acquire them, and how to generate code. The consistency and validation checks performed by these model-based tools are also present in DIDS-generated KA tools. The verification component of the MeKAs implements these checks.

The biggest drawback of DIDS is the large amount of time required to build a mechanism and MeKA library. Analysing a task, such as configuration design, to determine the necessary knowledge structures, mechanisms and MeKAs, requires a large amount of time (the analysis of configuration design presented by Balkany *et al.* (1993*a*) required one to two man-years of effort), numerous iterations, and a detailed understanding of existing knowledge systems that perform this task. In addition, once these constructs have been identified, significant investment is required to code and debug a set of process models capable of implementing the mechanisms efficiently.

Another limitation of DIDS is that we have not, to date, discovered a means for describing precisely the behavior and functionality of mechanisms and MeKAs. Precise descriptions would facilitate the reuse of mechanisms and MeKAs outside of the DIDS framework. They would enable other systems that reuse mechanism-like constructs, such as SBF, PROTÉGÉ II or KADS, to use DIDS mechanisms. We are experimenting with describing mechanisms and MeKAs using Ontolingua (Gruber, 1992; see Gruber, 1993, this issue), but the results of this experiment are inconclusive, at this point. We developed an ontology describing the knowledge structures and a few of our mechanisms, and are now in the process of porting the ontology to other user groups. From this experience, we will be able to more accurately access the portability issues.

Although most of our experiments have been with configuration tasks, our approach is not limited to configuration. Porting DIDS to new tasks requires that these new tasks be analysed to discover the knowledge structures, mechanisms, MeKAs, and process models that can be used to automate them. We are extending DIDS into new tasks, such as diagnosis and scheduling.

7. Summary

In this paper, we have presented an evolving methodology for building model-based knowledge-acquisition tools from pieces. These tools are constructed by assembling reusable software components, which we call MeKAs, that perform well-defined subtasks. The difficulty in building tools in this way is finding an ordering of MeKAs that will produce a meaningful dialog with the user. We believe that this problem can be overcome by exploiting dependencies in the knowledge structure assumed by the MeKAs, and by using a set of high-level strategies that are specific to design, the task we are exploring, but independent of a particular design task.

Our initial results have been encouraging. Several different configuration tasks

have been attempted. The knowledge structures were adequate to represent these problems, and the MeKAs were reusable. The strategies also proved helpful in creating the KAMs.

Alan Balkany and Iris Tommelein have made significant contributions to the DIDS research. In addition, the reviewers of this paper provided may valuable suggestions. This work was funded, in part, by the National Science Foundation Grant MIPS-905781 and by Digital Equipment Corporation. The views of the authors do not necessarily represent those of these funding organizations.

References

BALKANY, A., BIRMINGHAM, W. P. & TOMMELEIN, I. D. (1993*a*). An analysis of several configuration-design tools. *Artificial Intelligence in Engineering, Design, and Manufacturing,* in press.

BALKANY, A., BIRMINGHAM, W. P., MAXIM, B., RUNKEL, J. T. & TOMMELEIN, I. D. (1993*b*). DIDS: rapidly prototyping configuration design systems. *Journal of Intelligent Manufacturing,* in press.

BALKANY, A., BIRMINGHAM, W. P. & RUNKEL, J. T. (1993*c*). Solving Sisyphus by design. *Knowledge Acquisition,* in press.

BIRMINGHAM, W. P., GUPTA, A. & SIEWIOREK, D. (1992). *Automating the Design of Computer Systems: The Micon Project.* Boston: Jones and Barlett.

BIRMINGHAM, W. P. & KLINKER, G. (1993). Knowledge acquisition tools with explicit problem-solving methods. *The Knowledge Engineering Reviews,* **8,** 23–43.

BIRMINGHAM, W. P. & SIEWIOREK, D. P. (1989). Automated knowledge acquisition for a computer hardware synthesis system. *Knowledge Acquisition,* **1,** 321–340.

BIRMINGHAM, W. P. & TOMMELEIN, I. D. (1991). Towards a domain-independent synthesis system. In M. GREEN, Ed. *Knowledge Aided Design.* London: Academic Press.

BROWN, D. & CHANDRASEKARAN, B. (1987). *Design Problem Solving—Knowledge Structures and Control Strategies.* San Mateo, CA: Morgan Kaufman.

CHANDRASEKARAN, B. (1986). Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *AI Magazine,* Fall, 45–63.

ERIKSSON, H., SHAHAR, Y., TU, S., PUERTA, A. & MUSEN, M. (1992). *Task modeling with reusable problem-solving methods.* Knowledge Systems Laboratory, Stanford University, Report KSL-92-43.

GRUBER, T. R. (1992). *Ontolingua: a mechanism to support portable ontologies.* Stanford University, Knowledge Systems Laboratory, Technical Report KSL 91-66. Revision.

HAWORTH, M. S., BIRMINGHAM, W. P. & HAWORTH, D. E. (1992). *Optimal part selection.* University of Michigan, Computer Science and Electrical Engineering Division, CSE-TR-129-92.

HOROWITZ, E., KEMPER, A. & NARASIMHAN, B. (1985). A survey of application generators. *IEEE Software,* **2,** 137–156.

JOHNSON, M. V., Jr & HAYES-ROTH, B. (1988). *Learning to solve problems by analogy.* Stanford University, Department of Computer Science, Knowledge Systems Laboratory, Report No. KSL-88-01.

KLINKER, G., BHOLA, C., DALLEMAGNE, G., MARQUES, D. & MCDERMOTT, J. (1990). Usable and reusable programming constructs. *Proceedings of the 5th Knowledge Acquisition Workshop,* AAAI.

KRUEGER, C. (1989). *Models of reuse in software engineering.* Carnegie Mellon University, Computer Science Technical Report, CMU-CS-89-188.

LANGRANA, N., MITCHELL, T. & RAMACHANDRAN, N. (1986). Progress towards a knowledge-based aid for mechanical design. *Symposium on Integrated and Intelligent Manufacturing.* The American Society of Manufacturing Engineers.

MACKWORTH, A. K. (1987). Consistency in networks of relations. *Artificial Intelligence,* **8,** 99–118.

MAHER, M. (1987). Engineering design synthesis: a domain independent representation. *AI EDAM*, **3**, 22–38.

MARCUS, S. (1988). Salt: a knowledge acquisition tool for propose-and-revise systems. In S. MARCUS, Ed. *Automating Knowledge For Expert Systems*. Boston: Kluwer.

MARQUES, D., DALLEMANGE, G., KLINKER, G., MCDERMOTT, J. & TUNG, D. (1992). Easy programming: empowering people to build their own applications. *IEEE Expert*, **7**, 67–89.

MCDERMOTT, J. (1988). Preliminary steps toward a taxonomy of problem-solving methods. In S. MARCUS, Ed. *Automating Knowledge Acquisition for Expert Systems*. Boston: Kluwer.

MITTAL, S. & FRAYMAN, F. (1989). Towards a generic model of configuration tasks. *Proceedings of the 11th IJCAI*, pp. 1395–1401, Detroit MI, August.

NECHES, R., FIKES, R., FININ, T., GRUBER, T., PATIL, R., SENATOR, T. & SWARTOUT, W. (1991). Enabling technology for knowledge sharing, *AI Magazine*, Fall, **12**, 20–41.

PUERTA, A., TU, S. & MUSEN, M. (1992). *Modeling tasks with mechanisms*. Knowledge Systems Laboratory, Stanford University, Report KSL-92-30.

RICH, C. & WATERS, R. (1990). *The Programmer's Apprentice*. New York: Addison-Wesley.

RUNKEL, J. & BIRMINGHAM, W. (1992). Knowledge acquisition in the small. *Proceedings of the AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, October.

RUNKEL, J., BIRMINGHAM, W., DARR, T., MAXIM, B. & TOMMELEIN, I. (1992). Domain-independent design system: environment for rapid development of configuration design systems. In J. GERO, Ed. *Artificial Intelligence in Design '92*. Boston: Kluwer.

STEELS, L. (1990). Components of expertise. *AI Magazine*, **11**, 28–49.

WIELINGA, B. J., SCHREIBER, TH. A., BREUKER, J. A. (1992). A modelling approach to knowledge engineering. *Knowledge Acquisition*, **4**, 5–54.