Implementation of a Frame-based Representation In CLIPS

Hisham Assal and Leonard Myers[1]

Abstract:

Knowledge representation is one of the major concerns in expert systems. The representation of domain-specific knowledge should agree with the nature of the domain entities and their use in the real world. For example, architectural applications deal with objects and entities such as spaces, walls and windows. A natural way of representing these architectural entities is provided by frames.

This research explores the potential of using the expert system shell CLIPS, developed by NASA, to implement a frame-based representation that can accommodate architectural knowledge. These frames are similar but quit different from the 'template' construct in version 4.3 of CLIPS. Templates support only the grouping of related information and the assignment of default values to template fields. In addition to these features frames provide other capabilities including: definition of classes, inheritance between classes and subclasses, relation of objects of different classes with "has-a", association of methods (demons) of different types (standard and user-defined) to fields (slots), and creation of new fields at run-time.

This frame-based representation is implemented completely in CLIPS. No change to the source code is necessary.

Keywords:

Architecture, Design, Engineering, Expert Systems, Frames, Knowledge-Based System, Knowledge Representation.

Introduction:

Architectural design involves large amounts of information in often diverse fields of knowledge. In order to create a computer-aided design environment for architecture, there should be a uniform representation for architectural entities that is capable of describing all attributes and characteristics of these entities in different contexts of the design activity. Although architectural objects seem to be well defined as the components of a building such as space (a more generic term for room), wall or window, the attributes and characteristics of these objects vary in response to the context of the design activity. For example, in the conceptual design phase, a space may be described in terms of its orientation, adjacency to other spaces or/and access to circulation elements. Whereas in a different level of design, such as daylight analysis, the space may be described in terms of its geometry, window material or/and the amount of daylight it has. The knowledge representation scheme for such an environment should be flexible enough to handle the needs of different activities of design.

The ICADS Model:

The Intelligent Computer-Aided Design System (ICADS) is a project that is being developed in the CAD Research Unit of California Polytechnic State University, San Luis Obispo.

---

[1] Hisham Assal is a graduate student in the architecture department and Leonard Myers is a professor in the computer science department at the California Polytechnic State University, San Luis Obispo.

The ICADS model [1] provides on-line access to knowledge pertaining to the kind of design project under consideration; and expert assistance during the iterative analysis, synthesis and evaluation cycle of the design activity. It consists of several components that deal with architectural knowledge on different levels.

The first component is an existing drawing system that produces point/line drawings to represent the architectural solution. In order to allow for analysis or evaluation of the evolving design, there is a geometry interpreter [2] that transforms the point/line representation into architectural objects, such as spaces, walls or windows. The interpreter also formulates the relations that connect the objects (such as, the walls in a space) to provide a meaningful description of the evolving design solution. This information then flows to a control system (the blackboard) [3]. The blackboard receives different information from all the components of the system. It has knowledge about the information needed by every component and it uses this knowledge to efficiently propagate its information. The intelligent design tools (IDTs) are narrowly focused expert systems that perform the analysis and evaluation of the design and send their results back to the blackboard. If there is a conflict in the results of two or more IDTs, the blackboard tries to resolve it in the context of the project as a whole using its own set of rules (conflict resolver). There is also a relational database component that stores prototype information about building types and sites.

One of the inherent problems in this model is the diversity of the formats of information needed in different components. For example, the geometry interpreter produces architectural objects in C structure format, the database queries return tuples in SQL format and the IDTs use CLIPS facts. In fact this diversity is common in systems where a variety of databases are needed [4]. There is a need for a common representation to make it possible for all components to communicate with each other.

The common representation of information designed for the ICADS system is the frame-based scheme described in this paper.

CLIPS Knowledge Representation:

CLIPS is a forward chaining rule-based expert system shell, developed by NASA [5]. It has three major components:

- fact-list which is the working memory of facts.
- knowledge base which is the set of rules and initial facts.
- inference engine that controls the overall execution.

Information in a CLIPS expert system is represented in the form of facts. The structure of facts is quite simple. A fact is merely a list of one or more fields which may be one of three types: a word, a string or a number. A word is any field that does not start with a number or a special character; a string is any character or set of characters between quotes; and a number is always a floating point number. Fields cannot be lists themselves. That means that nested lists are not allowed in this environment. There is no restriction whatsoever on the field values that can be in a fact or the order of fields in a fact. In addition to the simple fact structure, there is a 'template' structure that was introduced in the CLIPS version 4.30. The 'template' provides two features: field identification and default values. The structure of a template has two components: a label and a list of name-value pairs. The use of field names in templates permits the fields to be identified regardless of the order in which they are written. It also makes it possible to provide default values for the fields declared in a template.

Templates enhance the representational power of facts in CLIPS. Further enhancement can be provided by a more general frame-based representation scheme.

The Frames Paradigm:

Frames provide a structured mechanism of representing different types of knowledge [6]. They have some powerful features that help to capture human knowledge in such a way as to facilitate both conceptual level and programming level uses of the knowledge. A frame can be viewed as a collection of information about an object. It may represent a physical object, such as window, or a conceptual object, such as climate. A frame may represent a class of objects by describing its general characteristics and relations to other objects. It may also represent an instance by specifying its class and specific characteristics. Classes may be arranged into taxonomies; i.e. a frame may represent a subclass which is a specialization of a class. The class information is available to any instance of the same class or of any of its subclasses through inheritance.

The structure of frames consists of slots that represent different types of information [7]. The content of a slot may be a value of any type (number, string, ... etc.); a restriction upon another slot's value (range, type, ... etc.); a demon, which is a method of performing a special task; a relation to another frame; or any other kind of information. Inheritance can be applied to any type of slot, or it can be suppressed for a particular instance. Different types of relations may be defined among frames, such as is-a, has-a, a-kind-of, ... etc.

Combining frame-based representation with pattern matching techniques adds power to frames in terms of reasoning facilities. Reasoning with frames involves several levels: class level, instance level and slot level. For example, operations may be performed on a particular slot in all instances of a class; a certain type of relation may be identified in all classes; and restrictions may be imposed on a type of value (e.g. boolean: true or false).

Creating Frames:

The implementation of frames in the CLIPS environment comprises three parts: representation, generation and manipulation.

- Representation is the form and collection of facts that compose a frame.
- Generation is the phase or module that creates new frames and/or slots and relations.
- Manipulation is the module that performs operations on frames, slots or relations, such as add, delete or modify the contents of a frame.

It should be noted that the manipulation rules are different from the application rules that use the information stored in frames without directly changing any of it. The basic purpose of the manipulation module is to provide a mechanism for dealing with frames so that the user can set up the conditions or restrictions or specify actions to be taken upon additions, deletions or changes in frame contents.

Representation of Frames:

A frame can hold either a class or an instance. If a frame holds a class, then the information in this frame will describe the basic characteristics of this class such as default values, demons as methods of obtaining values or performing particular tasks, names for the value slots in this class (without actual values), and relations between this class and other classes. It may also include any other information that the user wishes to have such as: restrictions on slot values, facets for describing

how to deal with a particular piece of information, ... etc. On the other hand, if a frame holds an instance, then the information in this frame will be the actual values for the value slots and the actual instance identifiers for the relations. Through inheritance, all the class information will be available to any frame of this class or any of its subclasses.

A frame is represented by a set of facts that have one or more common fields to connect them together. Each fact has a keyword in the first field to indicate the type of information it represents. The keywords are: CLASS, DEFAULT, DEMON, FRAME, RELATION, and VALUE. The second field has the class name which is used to connect all instances of this class, relate the class to its superclass, or establish a relation with another class. In the instance frames, there is a field for the frame identifier which is used to connect all the facts representing a particular frame instance. In addition to these basic fields, every fact contains different number of other fields to describe the piece of information it holds.

Definition of Classes:
The first step in creating frames is the definition of classes that will be used in the application. A class definition has the following components:

- A class header that declares the class name and its superclass (if any). If the class does not have a superclass (i.e. it is the uppermost level class), the class name is repeated in place of the superclass. The class header is a fact of the form:

(CLASS <class> <superclass>)
where CLASS is a keyword,
<class> is the class identifier and
<superclass> is its superclass identifier.

Since this class header is the only place that has information about the class/superclass relationship, the names of all classes and superclasses must be unique.

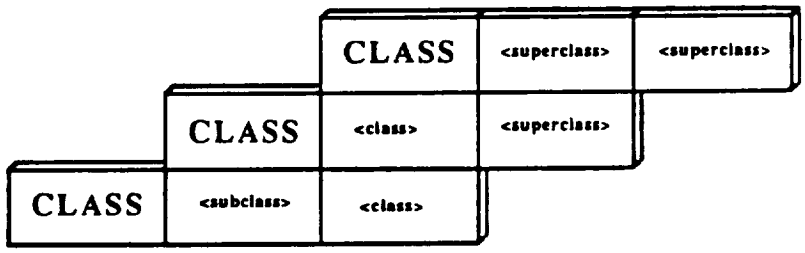

Fig. 1. Class Hierarchy.

- Default slots for all the default values in this class. A default value can be accessed by any instance of its class through inheritance. The default slot is a fact of the form:

(DEFAULT <class> <attribute> <value>)
where DEFAULT is a keyword,
<class> is the class identifier,
<attribute> is the slot name and
<value> is the default value.

- Demon slots that declare all the demons of this class as methods of obtaining values. A demon is represented by a fact of the form:

(DEMON <class> <attribute> <type>)
where DEMON is a keyword,
<class> is the class identifier,
<attribute> is the slot name that should have this value and
<type> is the type of the demon that controls its firing.

Along with this fact, there should be a set of one or more rules that actually describe the method of obtaining the value. Users can define the type of demon and set the conditions that control its firing. For example, if it is of type 'if-needed', it will fire only once when there is no current value for this attribute. However, it will not fire again until this value has been deleted. If it is of type 'if-changed', it will fire every time the value of this attribute has been changed. Since demons belong to classes, a fact must be asserted, when firing the demon, to indicate the instance of the class that will receive the result of the demon.This fact has the form:

(DEMON <class> <attribute> <instance> <type>)
where DEMON is a keyword,
<class> is the class identifier,
<attribute> is the slot name that should have this value,
<instance> is the frame identifier of the instance, and
<type> is the type of the demon that controls its firing.

- Value slots that declare the basic attributes of the class. These slots do not have values since the actual values will be in the instance frames. A value slot in the class definition is a fact of the form:

(VALUE <class> <attribute>)
where VALUE is a keyword,
<class> is the class identifier and
<attribute> is the slot name.

- Relation slots that describe the relation between this class and other classes. A relation in this implementation is a 'has-a' relation. As in value slots, relation slots do not have the actual instances of the classes. A relation slot is a fact of the form:

(RELATION <class> <other class>)
where RELATION is a keyword,
<class> is the class identifier and
<other class> is the identifier of the related class

The interpretation of this type of fact should be: every instance of <class> has an instance of <other class>. That means that the whole frame of <other class> is a part of the frame of <class>. However, a relation does not imply any inheritance. It is, rather, a way of defining the relationship between classes that are not derived from one another.

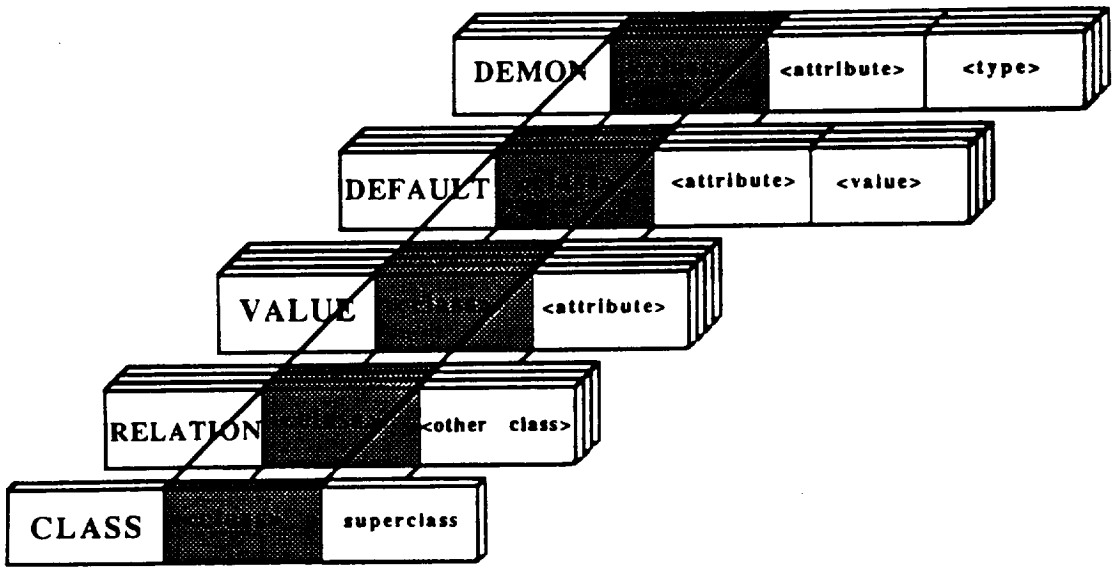

Fig. 2. Class Definition.

Definition of Instances:
An instance of a class is defined as follows:

- a frame is defined by a FRAME header which is a fact of the form
  (FRAME <class> <instance>)
  where FRAME is a keyword that should be in the first field,
  <class> is the name of the class of this frame and
  <instance> is the frame identifier.

The FRAME header may not be necessary in accessing the slot value in a frame, but it is useful in performing operations on the whole frame, such as displaying frame information, deleting a frame or relating a frame to another frame.

- a slot value is defined by a VALUE slot of the form:
  (VALUE <class> <attribute> <instance> <value>)
  where VALUE is a keyword,
  <class> and <instance> are the same as in the frame header,
  <attribute> is the slot name or attribute and
  <value> is the actual value of this slot.

The <value> field may be a single-field or a multi-field value depending on the nature of this slot. If the attribute in this slot has the nature of a list, such as the coordinates of a point (x,y), then a multi-field value should be used in the slot fact.

- a relation is defined by a RELATION slot of the form:
  (RELATION <class1> <class2> <instance1> <instance2>)
  where RELATION is a keyword,
  <class1> and <class2> are two class identifiers,
  <instance1> is an instance of class1 and
  <instance2> is an instance of class2.

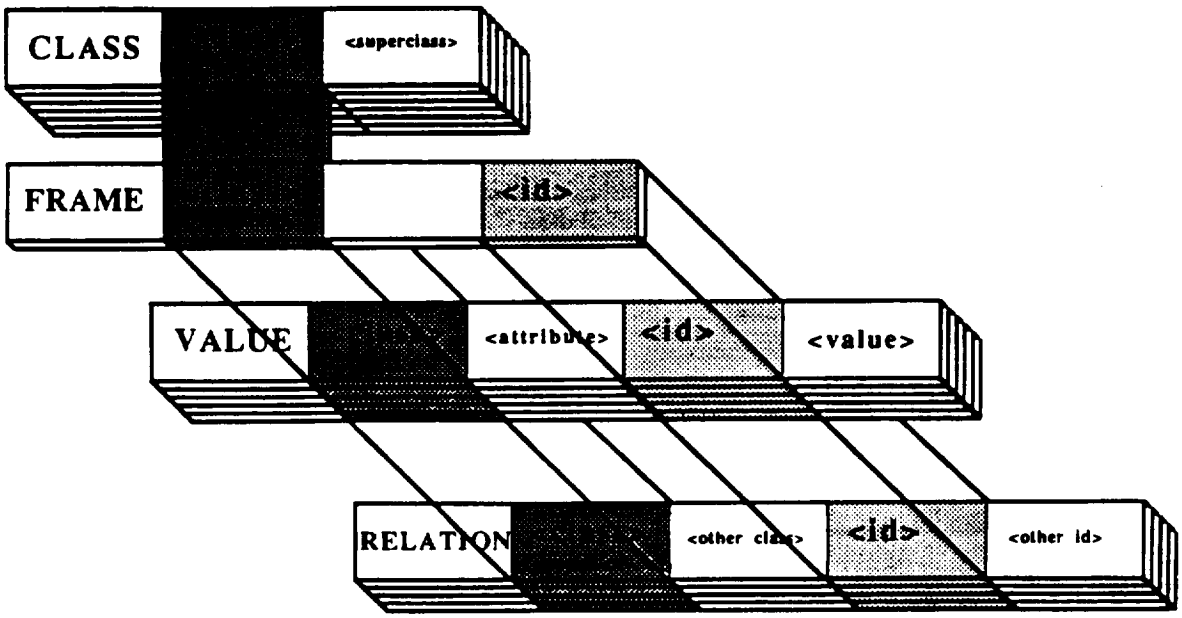

Fig. 3. Class-Instance Relation.

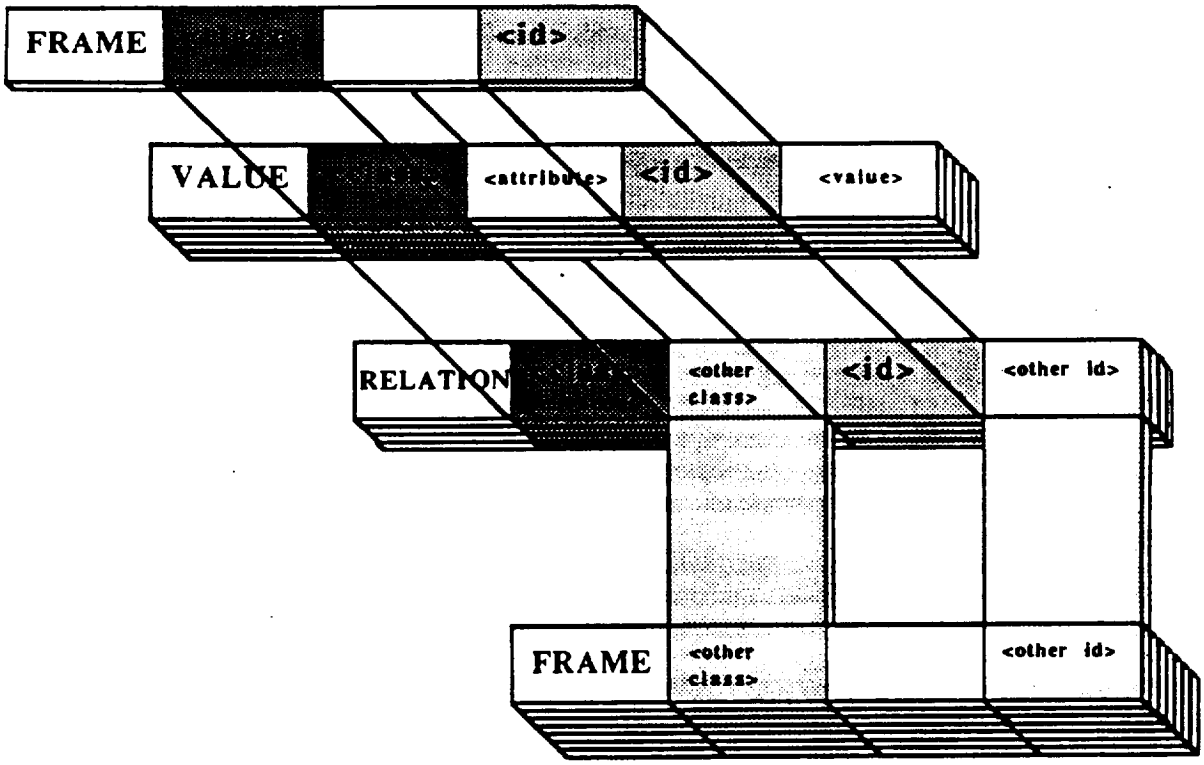

Fig. 4. Instance-Instance Relation (has-a).

Generation of Frames:
The definitions of the class frames are kept in a separate file to allow them to be reused in other programs. This file typically contains all the facts that describe each class and all the rules for the demons. The generation of instances for an application can be either static or dynamic. Static generation involves the creation of fact files that contain all the instances that are known prior to execution. Dynamic generation is usually achieved by having a module that is responsible for creating frames, slots or relations according to the state of the system and the conditions set by the user.

In the ICADS model, there are two modules that create frames dynamically: the Geometry Interpreter (GI) and the Attribute Loader (AL). The GI is responsible for creating frames that contain the geometry of the evolving solution drawn by the user in the CAD system . The AL is responsible for creating frames that contain the non-geometric attributes of the building being designed from a prototype database and all the relations that relate these frames to the geometric frames of the GI. The GI is a C module that was added to a modified version of CLIPS, while AL is a CLIPS module that has access to the SQL relational database.

Manipulation of Frames:
Frames are controlled by a module that takes care of performing the actions, enforcing the restrictions and checking the facets while manipulating the frames. The main three actions to be performed on frames are: ADD, DELETE and MODIFY. Each of these actions can be applied to FRAME, VALUE or RELATION slots (with the exception of MODIFY RELATION). If, for example, there is a restriction on a slot value to be of a certain type or within a certain range, then this module will check this restriction and enforce it.

Inheritance in Frames:
There is a set of rules that perform the inheritance operation. These rules are kept in a separate file that should be loaded with any application that uses inheritance. The inheritance rules have a priority (salience) of 10000 to allow the inheritance to take place as soon as it is invoked. The rules of the application itself should not have a higher priority.

Class-subclass inheritance:
Inheritance must be explicitly requested. This means that there should be a rule to issue a request for inheritance when the absence of a value is detected. The request is a fact that activates the inheritance rules. This fact has the form:

(INHERIT <class> <attribute> <instance>)
where INHERIT is a keyword,
<class> is the class name of the requesting frame,
<attribute> is the slot name to be inherited, and
<instance> is the requesting frame id.

When a request for inheritance is issued, the class frame of the requester is searched first for the requested slot. If it is found, its value is inherited; i.e. a VALUE slot is created for the requester with the value field. If the slot is not found and the class has a superclass, a request for inheritance is issued for the same slot in the superclass. This process continues until a slot with the required name is found or no other classes are to be searched. The slot to be inherited need not be in a VALUE slot. It may also be a DEFAULT slot or a DEMON slot. When a DEMON slot is inherited, the demon fires and creates a value. This value is then inherited in a VALUE slot.

Other Types of Inheritance:

Instances may implicitly inherit slots from other instances that are not in the same class hierarchy. In this case, DEMONs are used instead of the inheritance rules. Since DEMONs describe ways of obtaining values for specific slots, they can simply get the value of any other slot in the same instance frame or in any other frame. For example, if a wall instance has a slot for 'height', a space instance may get the value of this slot for its 'ceiling-height' slot using a DEMON in its class definition. This DEMON must have knowledge about the relationships between wall frames and space frames.

Reasoning With Frames:

The arrangement of frames as sets of separate facts connected by common fields makes it possible for different levels of reasoning to take place using the powerful pattern matching of CLIPS. Levels of reasoning involve:

* Class reasoning.
* Instance reasoning.
* Slot reasoning.
* Relation reasoning.

- Class reasoning: Using the class field in a frame, operations may be performed on all instances of this class. For example, to display the names of all spaces, a rule as the following may be used:

```
(defrule display-space-names
        (VALUE space name ?id ?value)
=>
        (fprintout t "Space " ?id " has the name " ?value crlf)
)
```

-Instance reasoning: Using both the class and the identifier fields, operations may be performed on all slots of a particular instance. For example, to display all the information of a particular wall instance 'wall-1', a rule as the following may be used:

```
(defrule display-wall-slots
        (VALUE wall ?attribute wall-1 $?value)
=>
        (fprintout t "The attribute " ?attribute " has the value(s) " $?value crlf)
)
```

- Slot reasoning: Operations may be performed on slots that have specific characteristics such as the name, the value or the number of values regardless of what frame they belong to. For example, to display the height of all the objects that have a 'height' slot, a rule as the following may be used:

```
(defrule display-heights
        (VALUE ? height ? ?value)
=>
        (fprintout t "The height of " ?object " " ?id " is " ?value " ft." crlf)
)
```

- Relation reasoning: The information in a frame that is related to another frame can be accessed by using the <other class> and <other id> fields in the relation slot. For example,

to display the length of all the walls in a space instance "space-1", a rule as the following may be used:

```
(defrule display-wall-length
        (FRAME space space-1)
        (RELATION space wall space-1 ?wall-id)
        (VALUE wall length ?wall-id ?value)
=>
        (fprintout t "Wall " ?wall-id " has length " ?value crlf)
)
```

Conclusion:

Rules are useful in representing knowledge about situations in the domain world and actions to be taken in each situation. In addition, there is also a need to represent the entities of the domain world, relationships among these entities, and operations that could be performed on them. These entities are referred to as objects. When dealing with a problem that uses objects, it is appropriate to use frames. This frame-based representation takes advantage of the pattern matching technique of CLIPS to provide a flexible yet powerful frame environment.

Flexibility is achieved by arranging the frame as a set of facts. This provides the ability to add a new slot at run time, deal with one slot in a frame without having to retrieve the whole frame, or remove a slot or modify its value without affecting the rest of the frame.

The power of this representation is attributed to the pattern matching, which allows different kinds of associations, such as class-subclass, class-instance, or class-class relations. Class-subclass relations are necessary in order to provide an effective taxonomy of the architectural entities in the ICADS system. Class-instance relations are used to effect the inheritance functions that make it possible to efficiently store the large numbers of architectural details necessary in the ICADS project. The class-class relation 'has-a' is used to synthesize, or define an object by specifying its components or features. The use of these associations in the prototype ICADS system has proved to be paramount to providing a robust, efficient representation of the architectural objects that naturally reflects the way the objects are perceived by human designers/architects.

Pattern matching also provides different levels of reasoning, such as, class reasoning, slot reasoning, or relation reasoning. Demons represent methods of performing operations that are specific to a class of frames. The frame manipulation module offers a means of control for the user (expert system developer) to impose some restrictions or to perform some tasks upon adding, deleting, or modifying slots.

REFERENCES:

1. Pohl, J., Chapman, A., Cotton J. and Myers L. *ICADS: Working Model Version 1, technical report, CADRU-03-89*, CAD Research Unit, Design Institute, Cal Poly, San Luis Obispo, CA, 1989.

2. Taylor, J. and Pohl, J. *A Geometry Interpreter for Extracting Architectural Objects from the Point/Line Schema of a CAD database*. Proc. Intersymp-90, Baden-Baden, West-Germany, August 6-12, 1990.

3. Taylor, J. *A Framework for Multiple Cooperating Agents in an Intelligent Computer-Aided Design Environment*. (Master Thesis). School of Architecture and Environmental Design, Cal Poly, San Luis Obispo, CA. 1990.

4. Howard, H.C. and Rehak, D.R. *KADBASE: Interfacing Expert Systems with Databases*. IEEE Expert, Fall 1989.

5. NASA. *CLIPS Architecture Manual (Version 4.3)*. Artificial Intelligence Section, Lyndon B. Johnson Space Center, NASA, May, 1989.

6. Minsky, M. L. *A Framework for Representing Knowledge*. In P. Winston (Ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975

7. Fikes, R. and Kehler, T. *The Role of Frame-Based Representation in Reasoning*. Communications of the ACM, vol. 28, No. 9, September 1985.