Contents

1	Introduction	1
2	Forms	3
3	The Description System	7
3.1	The Shell	7
3.2	The Knowledge Base	9
3.3	The Description	10
4	Features of the Description System	11
4.1	Data-Directed Computations	11
4.2	Data Integrity	12
4.2.1	Dependency Links	12
4.2.2	Change Propagation	13
4.3	Default Values	14
4.4	User Interface	15
4.4.1	Forms and Instances	15
4.4.2	Menus	16
5	Example	16
6	Conclusion	23
7	Acknowledgement	24

Developing Design Solutions Using the Frames Technique of Knowledge Representation

Timothy R. Wyatt, Department of Civil Engineering
North Carolina State University, Raleigh, NC, USA 27695

Neal Holtz, Department of Civil Engineering
Carleton University, Ottawa, Ontario, Canada K1S 5B6

William J. Rasdorf, Department of Civil Engineering
North Carolina State University, Raleigh, NC, USA 27695

Abstract

With computer aids becoming more prevalent in the design process, the problem arises as to how best represent the design artefact for both the computer tool and the design engineer. This paper introduces the frames technique, an artificial intelligence concept, and examines its suitability as a solution to this design engineering problem. The frames technique was selected over other representational formalisms because it resides at the conceptual modeling level, and, therefore, can represent a wide range of data types. This paper contains a background discussion on frames theory, based on Winston's model, and presents a controlled experimental study of the frames technique for the design of a conceptual model, using an application from the architecture/engineering domain.

1 Introduction

Engineering is the science of predicting the performance of a physical artefact based on its description. The part of the design process which directly involves the engineer is itself comprised of two separate procedures, as illustrated in Figure 1. The first, *synthesis*, is the creation of a description of a design artefact which potentially satisfies a goal set of performance criteria. The second, *analysis*, computes the actual performance of the described artefact. If the results of the analysis determine that the proposed solution does not satisfy all performance criteria, the results can be used in subsequent rounds of synthesis. Design is thus a potentially iterative process, evolving through successive cycles of synthesis and analysis.

Description, being central to the design process, deserves serious attention. In particular, any attempt to utilize computer aids to automate or partially automate the design process must resolve the problem of representing the design artefact in a manner that is suited both to computer manipulation and to engineering analysis, and subsequently, to design. The representation must provide a wide variety of data types and permit non-standard design solutions.

One major disadvantage of the usual database approach to design description is that the abstractions which may be used to describe the design artefact are implicit in the data model; procedures related to the artefact are not explicitly represented. However, applying the knowledge representa-

Figure 1: The design process.

Figure 2: The modeling process.

tion technique of *frames* to design description avoids this disadvantage by providing for the explicit definition of the entire set of abstractions which may be used in the description.

The frames approach, one of several techniques that have been developed for representing knowledge, expresses knowledge as an aspect of the representation of an object. It is commonly used in the artificial intelligence domain as a conceptual modeling method. It is our view that a conceptual model should be developed before a computational model or a computer model; the latter two should be systematically derived from the conceptual model (see Figure 2) [Rasdorf92]. With the frames technique, *knowledge* in the domain of description is associated with the typical objects of description. Knowledge consists of data and procedures which are attached to each object. An application-independent program, at the computer level, acts on this knowledge to aid the user in creating and interpreting descriptions. Thus, the frames method was a logical choice for our system, based on its knowledge representation capabilities and its conceptual fit with the level of representation we wished to address. The frames approach provides an important model for representing design artefacts and the knowledge associated with them.

In general, the design process is not standardized, meaning that for any design problem, a large number of solutions may satisfy the design criteria. In describing design artefacts, the frame-based description system can respond to the need to represent a wide range of data types. A description is in terms of explicitly defined *classes* of design objects. The definitions of these classes form a *knowledge base*. Because knowledge is organized into classes, any single design description may select

from a large number of available classes. Because knowledge—being associated with specific classes—is modular, the knowledge base can be built upon and use existing classes. Design knowledge can furthermore be associated with the appropriate level of detail. It is possible to modify or augment specific classes independent of the classes which may interact with them.

Frames have some interesting features which have been selectively exploited in various implementations. One property of frames that has proved of considerable interest to others, and is of central interest herein, is that of providing an object-oriented representation [Bobrow86]. While message sending is indeed a common feature of object-oriented languages, it is not an essential feature of object-oriented representations [Tichy87]. Another interesting frame feature results from the data and procedures both being attached in most frame systems to the frame slots. This allows each of these frame attributes to be more richly defined. An additional benefit is that procedures can be implemented as *demons*. A demon is a procedure that is invoked (and an action that subsequently occurs) as a side effect of some other action. In the FRL frames language, for example, the slots have *If-added*, *If-removed*, and *If-needed* demons [Goldstein77]. These procedures are not directly invoked. The *If-added* and *If-removed* procedures are run as side-effects whenever a value is added to or removed from the slot. The *If-needed* procedure computes a slot value as a side-effect of a request for the value. Another noteworthy feature of frames involves *matching*, which allows the system to arrive at a plausible complete description that satisfies partial data. Frames can represent a class of objects (or a stereotypical object). Faced with an unclassified object, one can classify it by finding its matching frame.

This paper intends to demonstrate that a modified version of Winston's implementation of the frames technique of knowledge representation is a suitable medium for representing design artefacts. Topics discussed include: 1) forms; 2) the frames-based description system; and, 3) features of the description system, including the system's user interface. An example application from the architecture/engineering domain is presented to demonstrate the capabilities of the description system.

2 Forms

Our description system [Glass93a] [Glass93b] was based on *forms*, a development of frames. The following frame facets are thus provided: *Value*, *Default*, *If-needed*, *If-added*, and *If-removed*. Each represents one aspect of the knowledge associated with the slot:

- The *Value* facet takes the actual value of the slot; i.e., the slot value or slot filler.
- The *Default* facet takes a default value which is used if the *Value* facet is empty.
- The *If-needed* facet holds a procedure for computing the slot value.
- The *If-added* facet holds a procedure that is implemented if a value is added to the slot.
- The *If-removed* facet holds a procedure that is implemented if the value is removed.

Figure 3: The layout of a simple frame.

The last three facets are referred to as demons because they are initiated not by a direct command but as a side effect of another procedure. Thus, a request for a slot value may trigger the *If-needed* procedure which computes the value. The storing of this value may in turn trigger the *If-added* procedure. Both procedures may request or change the values of slots in other frames and thus arouse other demons. The layout of this kind of simple frame is shown in Figure 3, and the layout of a form is shown in Figure 4.

The form organization used in the description system we are proposing contains extensions or modifications to the typical frames implementation. This modified implementation introduces features which are valuable for design description—some of which are available in other frames systems and others which are unique to our system. For our implementation, the following additional features were desired:

1. Differentiation between *classes* and *instances* of a class;
2. Specialization of local forms; and
3. Active integrity maintenance of data.

One of the key modifications made is the separation of instances and classes. This modification, which is a common feature of most frames systems, allows for the independent definition of classes and their instances. Thus, an independent knowledge base of class definitions can be compiled. An independent knowledge base is essential to engineering design since the same knowledge must be used for many different projects. The AKO relation typically has the dual meaning of “a sub-class of” and “an instance of.” This is not an unreasonable implementation. A sub-class constrains one or more parameters of the parent, and an instance constrains all parameters. An instance,

therefore, can be viewed as a sub-class with all parameters constrained. However, in the extended implementation, the AKO relation will mean exclusively "a sub-class of" (see Figure 4), while a new AIO relation will be introduced for "an instance of" (see Figure 6).

Another important extension of the frames model, not found in other frames systems, is the definition of local forms (see Figure 4). A local form is a class definition that is defined within the computing environment of another class and is thus accessible to instances of that class. The definition of local forms allows the nesting of computing environments and makes possible "block structure" programming within frames for a more perspicuous organization of knowledge. The local form can also be used to customize classes for special applications.

Included in this specialization is an uncommon rule of inheritance of super-class slots by the sub-class. While in most systems, including the most successful, the sub-class may inherit from multiple super-classes and may also override super-class slots, this kind of inheritance has disadvantages for design description. It must be recognized that under such a rule the sub-class is not a true sub-class of its super-class. The inheritance mechanism serves as a means of "elision" [Stefik86], that is, a means of eliminating redundant code by sharing code among similar objects. The specialization within a description, however, must express a more semantically rigorous relation such as that propounded by Brachman [Brachman85a] [Brachman85b]. It is important in description that instances of highly specialized classes be reducible to instances of more general classes. For this reduction to be possible, the sub-class must not only inherit the properties of its super-class but the constraints of the super-class as well. An important additional benefit is that because the super-class is richer in knowledge than its sub-classes, the knowledge base must be structured so as to include general classes containing the most general and widely applicable knowledge as well as sub-classes which amplify knowledge of special cases.

Our description system also introduces facets which are useful in design description but not commonly implemented in frames systems. It is especially important in engineering design to actively maintain the integrity of design data. Throughout the process of design, a description will undergo periodic revision. It is absolutely imperative that dependent data values are kept current and accurate. In most frames systems, instances will represent individual objects within an interacting system of objects. In such cases, integrity is maintained through the *If-added* and *If-removed* facets. When a new object is introduced to the system, it is "connected" to those objects with which it interacts. The *If-added* and *If-removed* facets serve to inform connected objects to update when slot values are changed. In a design description system, this type of integrity maintenance system is not sufficient. Data is not always exchanged between discrete system objects. Rather, what is required is a direct integrity link between the dependent slot and its ingredients (see Figure 5). By directly linking slots instead of objects, it is possible to implement an efficient integrity maintenance facility which will ensure that all dependent data are updated when ingredient data values are changed. Such a facility has been implemented.

Included in our method of maintaining data integrity is an uncommon means of handling default values. Commonly, a default value is an initial value which the user has the option of changing. In many cases, such defaults can supply the user with typical or suggested values of description

parameters. The disadvantage of this type of default is that it does not give the designer the additional option of entering "I don't know" as a parameter value. In the early stages of design, the values of many parameters are not known, and that they are not known should be stated explicitly. Inclusion of this third option in the description system makes possible a complete change in the meaning of the default value. The default value can now be used as a temporary filler for the input value. It will be non-authoritative, and values derived from it will similarly be non-authoritative. Our description system, in implementing this kind of non-authoritative default, treats default values as a separate system of values with their own value-generating procedures. This permits dependent values to be computed based on default parameters, but it also distinguishes these inferior values from values based on authoritative input values.

3 The Description System

This section discusses the description system, which is shown in Figure 5. The description system is comprised of three separate levels of program abstraction:

- The system shell is an application-independent program which, in effect, defines a language for the creation of knowledge bases and descriptions.
- The knowledge base defines the typical objects of description, thus defining the domain of description.
- A description itself is made up of instances of the typical objects defined in the knowledge base.

The user has access to the knowledge base and the description. There are in fact two levels of user: 1) the user who helps create the knowledge base, and 2) the user who builds descriptions for design using an existing knowledge base. The following subsections discuss the three components of the description system.

3.1 The Shell

The system *shell* is the description system program minus any knowledge of specific forms or instances. The shell, so called because it is empty of domain knowledge, is the engine that runs the description system. The shell interprets the forms and instances and performs the actual evaluations required to create and maintain the description. The shell program in the current implementation is written in Scheme, a dialect of Lisp [Scheme87a] [Scheme87b]. Lisp represents both data and procedures as lists. This feature allows procedures to be manipulated as data. Thus, the slot methods in the forms can be examined, modified and treated by the shell program as if they were data, and they can also be incorporated into computations performed by the shell and evaluated as procedures. The attached procedures in the form direct the course of computations by the shell.

Figure 5: The system architecture.

The shell defines a set of Lisp functions that can be used within forms to initiate actions such as the slot query function which allows a slot to query another slot for its value. The functionality of the shell is much like that of a database management system. Forms define classes of computational objects and their interactions. The shell provides the programming language in which the forms are written. Although the shell establishes the format of the form definitions and is able to interpret the forms, the shell has no knowledge of actual forms. Thus, any number of knowledge bases can be created to be run by the shell since the shell and the knowledge base are independent entities.

3.2 The Knowledge Base

The knowledge base defines the typical objects of description, thus defining the domain of description. This is accomplished by using forms. A form, representing the essential attributes of a class of objects, may be understood as a class definition. Objects within a class have common attributes. For example, all objects belonging to the class of beams have attributes of length, width, area, weight, etc. Every object in this class will give different values to the common attributes, and thus will be one of an infinite number of unique instances of the class. Using the technique of frames, attributes (simple data) are represented by the frame slots, and the knowledge associated with those attributes (procedures) are held in the slot facets (i.e., *Value*, *Menu*, *Default*, etc.).

Every form will have two kinds of attributes; *parameters* and *properties*. Parameters are the set of attributes that are sufficient to fully define any instance of the object class. The values of all other attributes will depend in whole or in part on the values of the parameters. In the case of the class of beams, one may consider the attributes of length and width as parameters of the form. Properties are attributes that are computed from the parameter values of the form. Thus, the parameters are the smallest set of attributes necessary to define an instance of a form, and the properties are attributes that can be derived from these.

Forms may also be defined as *aggregations* or *specializations* of already-existing forms. An aggregation is a class or object defined as a union or assembly of separately defined classes or objects. For example, a wide-flange shape may be an aggregation of three rectangles. A specialization is a definition of a form as a kind of existing form with additional constraints put on its parameter values. The relationship created by specialization is the AKO ("a kind of") relation: that is, the new form is a kind of existing form. It is the AKO relation that makes it possible for a specialization, or sub-class, to inherit the properties of its super-class form as well as the means to compute the value of its property attributes.

The form extends the frame in several ways. First, the form contains a special slot for the AKO relation. This relation enables slots and procedures to be shared between frames. It allows the super-class (parent frame) of a sub-class (child frame) to be specified. For example, the form *Beam* may be a kind of the form *Building Structure*. Second, the form has a *Defn* facet that holds a short textual definition of the slot for use by the explanation facility, which is discussed in Section 4.4. Unlike the frame, the form is broken into three parts: slots, local forms, and methods. The slots portion of the form is similar to that of frames except the *Value* facet may not only contain a

simple value, but also a procedure to compute the value, the name of a slot method to compute the value, or the name of a form.

The **Menu** facet, which holds a list of the classes of acceptable slot fillers, has also been added to a form. Thus, the slot **Cross-section** in the form **Beam** may have the **Menu** facet "(\$menu shape-2d)" to limit the slot value to instances of the form **Shape-2d**. The **Default** facet has been extended just like the **Value** facet to hold not only a simple value, but also a procedure, a method, or a local form. The facets **If-added** and **If-removed** function the same as those described previously for a frame. The second portion of the form-local forms-contains all the local forms that are associated with the form. The third portion-methods-contains the names of all the methods used by the form.

3.3 The Description

Since description is the focus of design, attaching knowledge to the object of description is a means of integrating the design process. This simple idea, that knowledge should be attached to the object of description, is the basic tenet of the frames approach to knowledge representation. A description, as a collection of objects, is also a collection of knowledge about those objects. Endowed with knowledge, the description becomes a model of the described artefact; that is, rather than being interpreted by external processes, the description is able to interpret itself.

Description is a simple process of filling the parameter slots of a form. Each slot value is the result of a design decision or a constraint that limits the range of possible interpretations of the description. A slot filler may be either a simple value or an instance name. When a slot is filled with an instance name, the slot is not fully constrained. The slot filler will have parameters that must also be constrained. A description, therefore, takes the shape of a hierarchy of nested instances. Thus, related instances together form a complete description.

The layout of an instance is shown in Figure 6. The organization of the instance mirrors that of the form. It too has the **Defn** facet that holds a textual definition of the instance. However, instead of the **AKO** relation, the instance has an **AIO** ("an instance of") relation. For example, the beam **B1** is an instance of the class **Beam**. Unlike a form, the slots of an instance contain values. For example, the form **Beam** may have a slot **Area** whose filler is a method to calculate the value of the area. The instance **B1** would have an actual value filling this slot, such as "96 square inches." Each slot value must have knowledge of both its *ingredients* and its *dependents*. An ingredient slot stores the names of its dependents in the instance facet **Cards**. For example, both the **Height** and **Width** slots would have the **Area** slot as one of their dependents. A dependent slot stores the names of its ingredients in the instance facet **Ingreds**. For example, the ingredients of the **Area** slot would be the **Height** and **Width** slots of the form **Beam**.

Figure 6: The layout of the instance.

4 Features of the Description System

This section describes the main features of the design description system. The system modifies the basic frames model to better meet the specific needs of describing artefacts for design. In particular, the frame system implementation includes 1) data-directed computations for value generation, 2) the dynamic maintenance of data integrity, and 3) a system of default values which are generated and maintained separately from input values. These features are described below. Also discussed is the system's user interface, an important component of any implementation.

4.1 Data-Directed Computations

The manner of property slot evaluation is both *object-oriented* and *data-directed*. Computation in the description system is based on an object-oriented approach to programming. This involves constructing a computational object corresponding to each object in the system to be modeled. Forms fulfill the requirements of a computational object in that they have both local state (i.e., local variables) and local process: local state is provided by the slots, and local process is provided by the methods. The slot values and methods are encapsulated within the form and are accessible only within the form. To gain access to values in other instances of forms requires that a message be sent. In our description system, the messages are names of slots whose values are to be returned. Each property slot has a value method which is evaluated as a *demon* when the value of the slot is requested. A demon is a procedure that is performed as a side effect of some other action. Computations in the description system are strictly data-directed. That is, a request for a value causes an evaluation, rather than a request for an evaluation causing the generation of a value.

The description system incorporates two modes of evaluation: *forward chaining* and *backward chaining*. Forward chaining starts with an initial set of facts and derives dependent facts. It requires the user to input basic ingredient data before the system can start its evaluation of dependents. Parameter slots are evaluated in the forward chaining mode; that is, all parameter slots of an instance are evaluated when that instance is created. Backward chaining starts with a *goal*—a fact that is to be proved—and works backwards, evaluating those rules that contribute to proving the goal. It requires that the user set up certain unevaluated data as goals of the evaluation. Property slots are evaluated in the backward chaining mode; that is, property slots are only computed when their value is requested.

4.2 Data Integrity

Integrity is “the maintaining of functionally related information so that the relations are satisfied” [Eastman78]. The integrity maintenance facility of the description system maintains pointers between functionally related data. These links form a dependency network along which data updates can propagate. Simply put, ingredients are used to determine dependents. Ingredients may be used to determine a single up-level dependent in the network or they may be used to determine multiple up-level dependents. In the case of the former hierarchical considerations govern the operation of the description system and in the case of the latter network considerations govern.

4.2.1 Dependency Links

The first duty of the integrity maintenance facility is to establish links between functionally related data. These dependency links must point both ways; any slot must have knowledge of both its ingredients and its dependents. The unit of integrity maintenance is the slot; hence, the procedures for integrity maintenance must be implemented at the slot level.

Since the computations in the description system are based on queries for ingredient slot values, i.e., a request for a value, the dependency links are established as side-effects of the slot queries. In the description system, dependent slots are evaluated only to satisfy a query from the user. Having unevaluated dependent values is not considered a breach of integrity. It may be that “area = height X width” but until the value of the slot **Area** is computed, the functional dependence is not recognized, and the values of the **Height** and **Width** slots can be changed without concern for integrity. However, once the functional relation has been established, integrity must be scrupulously maintained; that is, once **Area** has been calculated, one cannot arbitrarily change the values of its ingredients, **Height** and **Width**. If a querying slot fails in its self-evaluation and hence has no value, it does not have a functional relation with its ingredients and the integrity link need not be established; thus, if **Area** has not been calculated, the values of **Height** and **Width** can be freely changed. The integrity relation links slot values, not slots; if there is no value there can be no link.

While, in general, a dependency link is created only between a querying slot and its queried slot, i.e., between a dependent and its ingredients, there is at least one special case in which a

dependency link must be established with a slot other than the queried slot: when the queried slot does not exist. Valid queries can be directed only at known parameters and properties of an instance. However, it is possible that a valid query may be directed at a slot that does not exist. For example, the volume of a beam is computed as the beam's cross-sectional area multiplied by its length. The method for the slot **Volume** in the form **Beam** will query the slot **Cross-section** for a value of area. **Cross-section** is a slot containing possibly many facets, of which **Area** is one. If the slot **Cross-section** does not have a value, there is no slot **Area** to query. The slot **Area** will only exist when the slot **Cross-section** is filled. By linking the querying slot to the last existing slot along the access path—in this case the slot **Cross-section**—the querying slot (**Volume**) can be notified when this slot is filled and the as yet non-existent slot—the slot **Area**—is brought into existence. The description system establishes such links.

4.2.2 Change Propagation

The principal integrity operation is the deletion of a parameter value. When a slot value is deleted, the slot returns to its original unasked condition. Its many dependents no longer have valid values. There is no need to re-evaluate the dependents since they obviously will evaluate to a null value, when their value is again requested. Value deletion can easily be propagated along the dependency network. Deletion of a slot value causes the deletion of the whole network of its dependent slot values. But the user may delete only those values that the user has entered. The user cannot directly interfere with the value of property slots. These slots are given values in accord with the slot methods.

Although value deletion does not have to trigger slot re-evaluation, changing a value must do so. While a deleted slot will instruct its dependents to delete themselves, a changed slot will instruct its dependents to re-evaluate themselves. The notification process must operate in a breadth-first fashion. The immediate dependent slots should all be re-evaluated before any one of them passes on the change notice. As any one slot receiving an update notice has no way of knowing when it is okay to pass on the notice, it should actually receive two messages: the first message will tell it to update its value; the second will tell it to propagate the change. In turn, the slot must first inform all its dependents to update themselves, and then inform them all to propagate the change.

The value update process as implemented does propagate an inconsistency notice, but it does not directly propagate the update notice. Rather, it uses a *bulletin board* for storing update notices. The idea of the bulletin board is presented by Lafue as a means of delaying integrity checking [Lafue82]. Instead of sending an update notice to the dependent slot, the notice is posted on the bulletin board. Only those slots that do not have dependents need to be posted. By evaluating the last-changed slots posted on the bulletin board, the network of inconsistent slots can be re-evaluated using the normal backward-chaining mode of computing slot values.

One important advantage of not re-evaluating slots immediately is that if the user is performing many changes, re-evaluation will be suspended until all the changes are made. In this way, re-evaluation can be done once for all changes. Thus, if the user is making many changes to the data,

the bulletin board will be cleared only when all these changes are complete. It is not even necessary to force an update since the update will occur the next time an invalidated slot is queried. While the inconsistency notice is propagated down the dependency network, the update notice is propagated up the network.

4.3 Default Values

A default is "a failure to fulfill an obligation" [Oxford83]. The user of a program has a responsibility to supply all necessary input values. Should the user default on this responsibility, the program might carry on, nonetheless, with internally generated values, known as default values. They are important in civil engineering design description because the description will otherwise be incomplete until the end of the design process. It is often useful to anticipate completion of the design description with reasonable temporary values. These values, however, cannot be treated as input values. Their use is strictly limited to providing a semblance of completeness to the description.

Since the purpose of the default value is to replace a value the user has failed to provide, and since the user provides values only to parameter slots, default values may be reasonably limited to parameter slots. Given default parameter values, default values of all dependent slots could be evaluated using the slot value methods. However, the value method of a property slot may not always be appropriate for computing the default value. As default values may not necessarily be accurate, default methods may use less rigorous computations than the value methods, or they may rely on fewer ingredient data. There is every reason therefore to include in property slots default methods that are different from the value methods.

The default methods are attached, by reference, to the **Default** facet. As with the **Value** facet, there are several ways of indicating the default value: it may be a numerical value, a reference to an instance, or a method name. All the options available for designating the slot value are also available for the slot default, with one important exception: the default method cannot query the user. The alternative default methods are ordered as they are to be evaluated, with the more precise methods being evaluated first.

The previous discussion of integrity maintenance (in Section 4.2) did not consider the integrity of default values. Because of that, we would like to add here that the integrity maintenance system for defaults works much the same as it does for hard values, but must handle computations imposed by the use of alternative default methods.

A default query links the querying and queried slot via complementary pointers stored in the **Cards** and **Ingreeds** facets. The **Default** facet may contain several alternative default methods. Their names are listed in the facet in order of use. In processing a default query, the system will first try to retrieve a hard value. If a hard value cannot be evaluated, the system will try to evaluate the first default method. If this evaluation succeeds, the system will ignore the other default methods. If the first default method fails, the system will go on to try the next method, and so on until all alternative methods are exhausted. As each default method is evaluated, the

slot is linked to the ingredient slots of that default method. It may be that the different defaults have different ingredients; in fact this must be so, for if they had the same ingredients they would all succeed or fail together.

The goal of the integrity maintenance system is to maintain the best possible slot value. The links to ingredients of the failed queries are maintained so that if any of these ingredients should subsequently become available, the dependent slot can be updated. However, when the dependent slot is updated, it must remove all links to its previous ingredients. This prevents the update of an ingredient of a lower-ranked default, and instead causes the update of the value of a higher-ranked default.

4.4 User Interface

This subsection will look at the user interface of the description system, and an important part of that interface, the explanation facility. The user interface, essential and critical in any interactive computer program, serves as the medium that allows the user and the program to exchange information. The user interface presents a clear model of the workings of the description system to the user.

4.4.1 Forms and Instances

The most practical help the explanation facility can give the user is access to form definitions and instances. Displaying the forms is no problem since they are adequately defined in the knowledge base "as is." Accessing the forms is a matter of interactively displaying the definition that already exists in the knowledge base.

Instances are created cooperatively by the user and the description system. The display of an instance is limited to the display of its slot filler values. The values of the other facets are not displayed. Access to all of the other facets is possible by using the appropriate shell function. The display of instances, therefore, consists of slot names and their corresponding slot filler names. The slot filler value may be a numerical value, an instance, or a special filler. The special fillers are the values which have been tagged with an indication as to where they came from. For example, the slot filler value may contain "13-Default" indicating that the value, "13," was obtained from a **Default** facet. If the filler is a numerical value, the value is shown. If the filler is a global instance, the instance name alone is shown. Thus, instances are displayed as lists of attributes with values. Values that are themselves local instances are shown in their entirety but are indented to indicate their subordinate position in the description hierarchy. Defaults and other special fillers are also indicated as such.

4.4.2 Menus

The explanation facility assists the user in selecting slot values by providing explanatory text and a complete menu of allowable fillers. The **Menu** facet is the facet on the form which specifies these permissible values of the slot. To fill the parameter slots, and hence to complete the description, the description system queries the user for values. A query consists of a simple prompt for a value. The prompting for parameter values offers an excellent opportunity to include some in-context help to the user.

The **Menu** facet is a list of all the acceptable classes of fillers for the slot. The slot will not accept a filler that does not belong to the class of fillers specified in the **Menu** facet. Included in the menu are local forms defined in the context of the current slot and global instances.

In the menu of acceptable fillers, the hierarchy of fillers is indicated by progressive indentation, with the sub-classes being indented relative to their parents. Each item in the list is given a reference number. The user may select from the menu of fillers by entering the number, rather than by typing in the name. Also, each item is identified either as a form, a local form, or an instance.

As the names of the fillers may themselves be unfamiliar to the user, the user may call up a definition of any of the fillers in the menu. This is done by selecting the filler reference number. Each form and instance has an association that holds a textual definition that is displayed to the user upon request. Also, to make access to help as easy as possible, the **Ask-user** function that prompts the user for the parameter value also handles requests to the explanation facility. Therefore, requests to the explanation facility are made at the system prompt without interfering with the current interaction.

5 Example

This section applies the description system to the topic of building structures. An example description is presented that treats the structural frame as an assembly of independent elements with an explicit representation of member connectivity. The scope of the knowledge base in the following example is limited to the description of the main gravity-load-resisting elements of a common type of steel-framed industrial structure. This effectively limits the description to steel beams and columns, both of which can be represented as straight-line elements of constant cross-section. The need for the explicit representation of connectivity is apparent in the computations of building loads due to gravity. To establish load paths down through the structure, the topological relationships between the beams and columns must be established.

To illustrate the creation and use of a description, a two-story structure, whose floor and roof framing plans are shown in Figure 7, is described. The structure is defined by first establishing the elevations and column-lines, locating the foundation elements at column-line intersections, and inserting the columns and beams. Then, the loads on the foundations are calculated.

Figure 7: Floor and roof framing plans of two-story structure.

The following dialogue is from a session on the system in which the user is creating a description of the building shown in Figure 7. The system prompts are shown in regular typewriter font, and the user responses are underlined. The numbers shown in brackets, e.g. "[6]," represent selected portions of the user's overall dialogue with the system for this example. The reader can see by following the numbers the amount of interaction that is not shown here. Figure 8 illustrates the schematic description of the partial definition of the structure given in this example.

The first step in describing the structure is to define the elevations of the foundation, of the base, Floor1, and of the roof by providing the identifiers, the definitions, and the elevations of the foundation, floor, and roof. For example,

```
[5] (create-instance 'elevation)
Instance id: foundation-el
Instance defn (string): 'Elevation of Foundation'
Y: 1250
```

```
[6] (create-instance 'elevation)
Instance id: floor1-el
Instance defn (string): 'Elevation of Floor1'
Y: 6250
```

```
[7] (create-instance 'elevation)
Instance id: roof-el
Instance defn (string): 'Elevation of Roof'
Y: 10250
```

Next, the column-lines are defined. This involves entering the column-line identifier, a definition of the column-line, and a definition of the three-dimensional vector defining the line. Thus, column line L1 is a line of infinite length at a location of 0 mm from the building origin and running parallel to the z-axis. Angle of rotation is measured relative to the z-axis. For example,

```
[8] (create-instance 'column-line)
Instance id: L1
Instance defn (string): 'Column line L1'
POINT: vector-zx
X: 0
Z: 0
ROTATION: 0
```

```
[9] (create-instance 'column-line)
Instance id: L2
Instance defn (string): 'Column line L2'
POINT: vector-zx
```

Figure 8: The schematic description of a (partial) definition.


```

X: 7000
Z: 0
ROTATION: 0

[10] (create-instance 'column-line)
Instance id: LA
Instance defn (string): 'Column line LA'
POINT: vector-zx
X: 0
Z: 7000
ROTATION: 90

```

Since the locations of the building columns will be defined relative to their supporting foundation elements, it is the foundation elements that are located at the intersection of column-lines. In the following, foundation element F1 is located at the intersection of the column lines LA and L1.

```

[19] (create-instance 'foundation)
Instance id: F1
Instance defn (string): 'Foundation at base of column C1'
ELEVATION: foundation-el
LOCATION-ZX: column-line-intersection
COLUMN-LINE-1: LA
COLUMN-LINE-2: L1

```

The columns are connected to the foundation elements. In order to define a column, its foundation element and its top elevation must be entered. The following example shows how a column can be created, as well as how the explanation facility can aid in creating the description. In the dialogue the user types a "?" requesting help. The user has already defined six footings (F1 - F6). The allowable user response is indicated as one of those previously defined footings or the user may define other local instances, i.e., an instance of Beam, Column, or Foundation.

```

[22] (create-instance 'column)
Instance id: C1
Instance defn (string): 'Column C1'
COMPONENT: ?
**DEFN: Host component
***MENU***
1: BEAM form
2: COLUMN form
3: FOUNDATION form
4: F1 instance
5: F2 instance

```

```

6:      F3 instance
7:      F4 instance
8:      F5 instance
9:      F6 instance
COMPONENT:  F1
TOP-ELEVATION:  roof-el
CROSS-SECTION:  nil

[23] (create-instance 'column)
Instance id:  C2
Instance defn (string):  "'Column C2'"
COMPONENT:  ?
**DEFN:  Host component
***MENU***
1:  BEAM form
2:  COLUMN form
3:  FOUNDATION form
4:  F1 instance
5:  F2 instance
6:  F3 instance
7:  F4 instance
8:  F5 instance
9:  F6 instance
COMPONENT:  F1
TOP-ELEVATION:  roof-el
CROSS-SECTION:  nil

```

The beams are located relative to their supporting columns and beams. The description of beams includes the beam loading. The beam loading has been computed manually based on tributary areas and applied to the beams as uniformly distributed linear loads. In the following example, beam B1 is defined according to the two columns that support it, C1 and C2. It has two distributed loads, with values 16.67 and 17.5, on different portions of its length. There is a 16.67 N/mm load on its first 2000 mm of length and a load of 17.5 N/mm on its last 3000 mm of length. At this point of definition the cross-section and material are unspecified.

```

[31] (create-instance 'beam)
Instance id:  B1
Instance defn (string):  "'Beam B1'"
COMPONENT:  C1
ELEVATION:  floor1-el
COMPONENT:  C2
ELEVATION:  floor1-el

```

```

POINT-LOADS: nil
DISTRIBUTED-LOADS: lf-55
L1: 0
L2: 2000
DIST-LOAD: 16.67
DISTRIBUTED-LOADS: lf-55
L1: 4000
L2: 7000
DIST-LOAD: 17.5
DISTRIBUTED-LOADS: nil
CROSS-SECTION: nil
MATERIAL: nil

```

Finally, all the structure components are combined in an instance of the form **Structure** by entering all the column-lines, foundation elements, beams, and columns that make up the structure. Specifying a value of "nil" ends the entry of information and the system proceeds on to additional prompting.

```

[49] (create-instance 'bldg-structure)
Instance defn (string): 'Assembled structure'
COLUMN-LINES: L1
COLUMN-LINES: L2
COLUMN-LINES: LA
.
.
COLUMN-LINES: nil
FOUNDATIONS: F1
.
.
FOUNDATIONS: nil
COLUMNS: C1
COLUMNS: C2
.
.
COLUMNS: nil
BEAMS: B1
.
.
BEAMS: nil

```

The above description is incomplete because the cross-sections of the structure components have not been input. Nevertheless, default values of some important properties can be computed. Since

the computation of gravity loads does not depend on the properties of the component cross-sections, the structure loads can be computed as hard values.

Because connectivity is explicitly represented in the description, the loading on any component or connection can also be computed. The loads supported by the structure foundations are obtained as follows:

```
[58] (get-value 'F1 '(load *)) nil
248.941224489796
```

```
[59] (get-value 'F2 '(load *)) nil
728.088775510204
```

The integrity maintenance system dynamically updates the properties of this description (including the loading data) whenever an input value is changed by the user.

6 Conclusion

This paper described a specific implementation based on the conceptual frames model of knowledge representation. The model represents engineering design artefacts in terms of knowledge which includes both data and procedures. Thus, the knowledge associated with an object is inseparable from its representation.

Engineering knowledge is diffuse and extensive. To encapsulate the knowledge pertaining to even a very limited description domain presents a considerable challenge. Practically, one must recognize the inherent difficulties of representing such knowledge in a form that can be processed by a computer. In practice, the design process strongly tends to produce artefacts that are easy to describe and analyze. Knowledge is the limiting factor. Only an increase in knowledge can lead to a wider variety of practical design solutions. So, by investigating frames and finding them suitable for representing both data and procedures, we can with confidence say they can represent knowledge in the design domain. That is a step in the right direction.

The research described herein has been motivated by a desire to shed new light on some familiar issues in engineering design by applying somewhat unfamiliar computing techniques. The frames approach offers a model for design integration that recognizes that there are many aspects of knowledge. This model is able to accommodate the inevitable need for modification and extension of the robustness of the representation and scope of the domain of knowledge. Since description is the focus of design, attaching knowledge to the objects of description is a means of integrating the design process.

7 Acknowledgement

Thanks are due to Don Duchesne of Adjeleian Allen Rubeli Ltd., Ottawa, for kindly supplying the structural drawings which were used as the basis for the example description, and to the CASCADE (Center for Advanced Studies in Computer-Aided Design and Engineering) Organization for the use of their computing facilities. This work was partially supported by NSERC Operating Grant A1419.

References

- [Bobrow86] Bobrow, D.G., M.J. Stefk. "Perspectives on artificial intelligence programming." *Science*, 231:951-957, 1986.
- [Brachman85a] Brachman, R.J., and J.G. Schmolze. "An overview of the KL-ONE knowledge representation system." *Cognitive Sciences*, 9:2:171-216, 1985.
- [Brachman85b] Brachman, R.J. "I lied about the trees; or, defaults and definitions in knowledge representation." *The AI Magazine*, 6:3:80-93, 1985.
- [Eastman78] Eastman, C.M. "The representation of design problems and maintenance of their structure." *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, Latombe, ed., Amsterdam: North-Holland, 1978.
- [Glass93a] Glass, A., N.M. Holtz, and W.J. Rasdorf. "Features of frame representations for design artefacts." *International Journal of Engineering with Computers*, Springer Verlag, submitted for review, January 1993.
- [Glass93b] Glass, A., N.M. Holtz, and W.J. Rasdorf. "A system for describing design artefacts using the knowledge representation technique of frames." *International Journal of Engineering with Computers*, Springer Verlag, submitted for review, January 1993.
- [Goldstein77] Goldstein, I.P., and R.B. Roberts. "NUDGE, a knowledge-based scheduling program." *Fifth International Joint Conference on Artificial Intelligence (IJCAI-5)*, pp. 257-263, 1977.
- [Lafue82] Lafue, G.M.E. "Semantic integrity dependencies and delayed integrity checking." *Proceedings of the Eighth International Conference on Very Large Data Bases*, pp. 292-299, 1982.
- [Oxford83] *The Oxford Paperback Dictionary*, London: Oxford University Press, 1983.
- [Rasdorf92] Rasdorf, W.J., and O.Y. Abudayyeh. "NIAM conceptual database design in construction management." *Journal of Computing in Civil Engineering*, American Society of Civil Engineers, Vol. 6, No. 1, January 1992.
- [Scheme87a] Texas Instruments Inc. *PC Scheme User's Guide*. Publishing Center, Data Systems Group, Texas Instruments Inc., Austin, Texas, 1987.

- [Scheme87b] Texas Instruments Inc. *TI Scheme Language Reference Manual*. Publishing Center, Data Systems Group, Texas Instruments Inc., Austin, Texas, 1987.
- [Stefik86b] Stefik, M., D.G. Bobrow, and K.M. Kahn. "Integrating access-oriented programming into a multiparadigm environment." *IEEE Software*, 3:1:10-18, 1986.
- [Tichy87] Tichy, W.F. "What can software engineers learn from artificial intelligence?" *Computer*, 20:11:43-54, 1987.
- [Walker88] Walker, E.L., M. Herman, and T. Kanade. "A framework for representing and reasoning about three-dimensional objects for vision." *The AI Magazine*, 9:2:47-58, 1988.
- [Winston77] Winston, P.H. *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1977.
- [Winston81] Winston, P.H., and B. Horn. *LISP*. Reading, MA: Addison-Wesley, 1981.