A System for Describing Design Artefacts Using the Knowledge Representation Technique of Frames

Alan Glass, Senior Engineer

Monenco Agri Inc., Oakville, ON, Canada L6H 6A3

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

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. Recognizing these needs, we have devised a model, referred to as the description system, that is based on the frame technique of description or knowledge representation. Our goals were to explore the suitability of frames for modeling design artefacts, evaluate their characteristics, explore their robustness, and assess their applicability in the structural engineering domain. This paper reports on the development of a computer implementation of a system, referred to as a description system, designed to assist us in achieving these goals. The description system consists of three levels: the system shell, the knowledge base and instantiations of descriptions. This system implements various facilities which are useful to designers, including the computation of design values, integrity maintenance, providing default values, and supporting explanation. This paper presents an overview of the three levels of the description system, briefly discusses the main features of this system, and illustrates a civil engineering application involving the design of a two-story structure.

1 Introduction

Descriptions of the physical world are essential to explaining the behavior of its artefacts. Descriptions support mathematical, statistical, physical, and computational models. But what form should the description take and how should it be expressed? Our interest is in the development of computer models to aid in the automation or partial automation of the engineering design process. It is of

Figure 1: The modelling process.

interest to us, therefore, to investigate: the suitability of modeling tools, techniques, and methodologies; the match between the desired level of detail of representation and the representational mechanism itself; and, the strengths and weaknesses of the tools available to us at each level. In this paper we report on the evaluation of one such representational mechanism (frames) for use in developing computational descriptions of design artefacts.

Our view of the modelling process is presented in Figure 1. A domain model, in the language of the domain, is derived from the physical artefact. A conceptual model, examples of which include NIAM and EXPRESS, is used to make the transition from the domain model to the computational model [Rasdorf92]. In this paper we will use the frame technique as such a conceptual model.

The theory of frames was introduced by Minsky in a seminal paper [Minsky75]. Frames is one of several techniques that have been developed for representing knowledge. Unlike the most familiar of these techniques, Expert Systems—which represent knowledge as rules of inference that are processed by an independent “inference engine”—the frame technique expresses knowledge as an aspect of the representation of an object. It is commonly used in the artificial intelligence domain as a framework for computational model specification. Frames are linked together in semantic networks, but whereas semantic networks emphasize relationships, frames emphasize objects [Winston77]. Thus, the frame method was a logical choice as a basis for our system, because of its knowledge representation capabilities and its conceptual realm.

With the frame technique, knowledge in the domain of description is associated with the typical objects of description. Knowledge consists of data and procedures which are associated with each object. An application-independent program, at the implementation level, acts on this knowledge to aid the user in creating and interpreting descriptions.

This paper presents a system for describing design artefacts which was implemented using the knowledge representation technique of frames. The *Communications of the ACM* has devoted a special issue to the topic of knowledge representation [ACM86]. Our system can respond to the need for representing a wide variety of data types in the description of design artefacts. Description is in terms of explicitly defined classes of design objects. The definition of these classes forms a knowledge base. Because this knowledge is modular, the knowledge base can be used to create a variety of design descriptions and can be incrementally expanded to provide an even larger set of representational building blocks for design descriptions.

2 Overview of the Frames Technique

As mentioned above, the description system is implemented using the knowledge representation technique of frames. A frame is a data structure that consists of a number of slots (data fields). Each slot has its own list of properties which are referred to as facets, i.e., its own attached knowledge. The attached knowledge may include both data and procedures; for example, there may be a procedure to compute the slot value, there may be a default value, there may be a constraint on the slot value, there may be comments, etc. A frame can be used to describe an object, an event or any stereotypical situation. For example, the frame *Beam* may have the slot *Area* whose value is a pointer to the frame *Rectangle*.

The layout of a simple frame is shown in Figure 2. For a frame, the following facets are provided: *Value*, *Default*, *If-needed*, *If-added*, and *If-removed*. Each represents one aspect of knowledge associated with a slot. The *Value* facet takes the actual value of the slot. For example, the frame *Beam* may have a slot *Width* whose value is "12 inches". The *Default* facet takes a default value which is used if the *Value* facet is empty. For example, if no width is specified, then the slot may use a default value of 16 inches. In the FRL frames language, the slots have *If-added*, *If-removed* and *If-needed* demons [Goldstein77] which are initiated not by a direct command but as a side-effect of another procedure. The *If-needed* facet holds a procedure for computing the slot value. For example, if the frame *Beam* has a slot *Area*, then its *If-needed* facet may contain a procedure for computing the area of a beam. The *If-added* facet holds a procedure that is implemented if a value is added to the slot, whereas the *If-removed* facet holds a procedure that is implemented if the value is removed. Thus, a request for a slot value may trigger the *If-needed* procedure which

Figure 2: The layout of a simple frame.

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.

A fair number of frame languages have been developed since the concept of frames was introduced. Each was developed for a different application and thus exploits different aspects of the frame theory. Among the more important and interesting applications are KEE [Fikes85] and LOOPS [Steff86a] [Steff86b].

3 Scope

The purpose of this paper is to present a system in which the frame model of knowledge representation has been applied to represent engineering design artefacts in terms of the knowledge that is associated with them. The knowledge of objects is thus inseparable from their representation. We will show that the frame approach to design description, incorporating both data and procedures, has a number of advantages. One such advantage is the ability of the designer to expand or customise the knowledge base to suit a given project. This can be easily done because the typical objects of description are explicitly defined in an independent knowledge base.

We will also show how this system modifies the basic frame model to better meet the specific

needs of describing artefacts for design. In particular, theme system implemented includes data-directed computations for data value generation, the data maintenance of data integrity, constraints on the values of data items, and a system of the values which are generated and maintained separately from input values. Finally, this paper presents an engineering application of this system in order to provide some degree of demonstration that the frame model is a suitable medium for representing design artefacts, although we do not report that rigorous scientific testing was conducted to establish or validate a proof of this assertion.

4 The Description System

This section discusses the description system, which is shown in Figure 3. 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 design, 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 designs using an existing knowledge base. The following subsections discuss the three components of the description system.

4.1 The Shell

The system shell is the description system program minus 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 instantiates and performs the actual evaluations required to create and maintain the description. The shell runs in the current implementation is written in Scheme, a dialect of Lisp [Scheme79] [Scherb]. Lisp represents both data and procedures as lists. This feature allows procedures to be manipulated as data. Thus, the slot

Figure 3: The system architecture.

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 procedure in the form direct the course of computations by the shell.

The shell defines a set of Lisp functions that can be used within forms (described in Section 4.2.1) 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 these 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.

4.2 The Knowledge Base

The knowledge base defines the typical objects of description, thus defining the domain of description. This is accomplished by using forms which have layouts very similar to those of frames and which essentially encapsulate frames.

4.2.1 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 that 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.

4.2.2 Comparison of Forms to Frames

The layout of a form, shown in Figure 4, is very similar to the layout of a frame (see Figure 2). However, 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 6. 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.

Figure 4: The layout of the form.

Figure 5: The layout of the instance.

4.3 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 5. 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 *AKO* ("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 *Ingredients*. For example, the ingredients of the *Area* slot would be the *Height* and *Width* slots of the form *Beam*.

5 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.

5.1 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 [Ahmed91]. 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 processes: 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 the names of slots whose values are to be returned. Each property slot has as 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.

5.2 Data Integrity

This section will look at the role of integrity maintenance in the description system. 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.

5.2.1 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-action 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 no 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 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 queried slot: when the queried slot does not exist. Valid queries can be directed only at knowparameters 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* query the slot *Cross-section* for a value of *Area* (*Cross-section* is a slot containing properties of which *Area* is one.) If the slot *Cross-section* does not have a value, then the slot *Area* to query. The slot *Area* will only exist when the slot *Cross-section* is filled. By using 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 *Area* is brought into existence. The description system establishes such links.

5.2.2 Change Propagation

The principal integrity operation is the deletion of a parameter. 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 only 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 network of its dependent slot values. But the user may delete only those values that he has entered. The user cannot directly interfere with the value of property slots. These slot-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 Latue as a means of delaying integrity checking [Latue82]. 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.

5.3 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. This subsection

will examine the use of default values in the description system. The description system clearly distinguishes default values from parameter values derived from user input. Default values are evaluated by default methods in response to specific default queries.

5.3.1 Soft Defaults

Defaults are implemented as *soft* values; that is, they are clearly identified as being different from an input value. Dependent values that are derived from default values must also be considered as default values. Values, therefore, are of two kinds: hard values (derived from input values) and default values (derived in whole or in part from default values). Clearly, any request for a value must indicate whether a hard value or a soft value is to be returned. The kind of value that is requested will depend on the intended use of the value.

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. This is a very important concept and provides an automatic mechanism for encoding approximate design methods into the system along with exact design methods. The method selected to perform dependent value computations will be determined by the data itself. The point is that very reasonable design descriptions can be generated early in the design process and can be modified and molded to a clearer, and more accurate, design description as more accurate data becomes known.

Furthermore, it should always be possible to evaluate a default. However, the best defaults depend on input values which might or might not exist at any given time. In anticipation of this possibility, the description system accommodates alternative defaults. If the actual default method cannot be evaluated, a less accurate one, incorporating fewer input values, can be used in its place.

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.

5.3.2 Integrity Maintenance

The previous discussion of integrity maintenance (in Section 5.2) 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 **Ingredients** 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.

6 User Interface

This section 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.

The description system offers an interactive explanation facility whose purpose is to present the user with information whenever it is needed. The user may at any time call forth a display of the forms in the knowledge base or a display of any instance. The explanation facility offers some in-context help during the creation of a description which is composed of instances of forms. Also, at the prompt for a parameter value, the user may request some explanatory material concerning the slot value. This explanatory material includes a short textual definition of the slot, a menu of permissible slot values, and the default value of the slot. Therefore, the user may obtain the definition of any item presented in the menu or may select the slot value directly from the menu.

The conceptual model of the description system is the image of the form as a questionnaire with blanks to be filled in. Each individual instance in an overall description can be viewed as a separate questionnaire, one that can be used to fill a blank in another. The final description is a sheaf of these questionnaires with the blanks filled in; though each is a separate instance, together they form a single description hierarchy. Thus, the description is represented as a hierarchy of nested instances.

At the implementation level, the interaction between the user and the system is conducted through the exchange of queries. This style of interaction suits the data-driven style of computation employed by the description system.

6.1 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 values. 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.

6.2 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 have been requests to the explain facility. Therefore, requests to the explanation facility are made at the system prompt without interfering with the current interaction.

7 Example

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

7.1 Knowledge Base

In the knowledge base the class of physical objects is represented by the form `Object`, which has two sub-classes: `Component` and `Assembly`. `Component` is the class of discrete objects, having two parameters: `Geometry`, which encodes both shape and size, and `Material`. `Assembly` is the class of assembled objects. The parts of an assembly are discrete components, or may themselves be assemblies.

Two important specializations of the form `Component` are `Area` and `Beam`. These new forms constrain the `Geometry` and `Material` parameters of the form in terms of their own parameters. In both cases the slot `Geometry` is constrained to a specialization of the global form `SLEB` (straight-line extruded element). The form `SLEB` parameter slot `Line-vector` and `Cross-section`. The slot `Line-vector` defines a vector local three-dimensional space; this is the vector along which the cross-section of the shape is swept.

The form `Assembly` is defined as an aggregation of the former. The `Volume` and `Weight` slot values of the form `Assembly` are computed as the sum of the `vs` and `weights` of the parts of the

assembly being described. The solids modeller chosen to work with the knowledge base is PADL-2 [Hartquist83], a constructive solids geometry (CSG) modeller. The CSG representation of an assembly is a compound of the CSG representations of the elements. The function `Padl2-assembly` takes a list of individual CSG representations and returns the representation of the assembly.

To fully use the integrity maintenance system to manage changes in the description, the locations of the parts of the assembly are given in terms of relative geometry. Beams, for example, are located relative to the columns or beams by which they are supported. These relative locations can be maintained when the locations of objects are changed. Thus, moving a column causes the elements supported by that column to be moved correspondingly.

The form `Elevation` is used to locate both beams and columns. It specifies a horizontal plane in terms of its y -axis intercept. In a building description, columns are usually located relative to a set of building column lines. The form `Column-Line` is defined as `AKO` the form `Horizontal-Line`. The relative locations of the parts of the assembly are given in terms of specializations of the form `Vector-3d`. These include the forms `Column-Location`, `Column-Line-Intersection`, `Beam-Location`, etc. The location of any element can be defined relative to its supporting element.

The knowledge base defines connectivity in such a way that a component does not directly connect to another component. Rather, the connection is mediated by a separate entity known as the `Connection` form; components are supported by connections, and connections are supported by components. For example, if beam `B1` is supported by column `C1`, a single connection will support `B1` and be supported by `C1`. A `Beam` or `Column` form has two kinds of slots that link the component to connections: slots for the supporting connections and those for supported connections. The supporting connections of a component are fixed both in number and location. For example, a column is supported by a single connection at its base; all of the gravity load supported by the column is transferred through this single connection. A simply-supported beam is supported by exactly two connections, one at either end of the beam, and the gravity load on the beam is distributed between these two supporting connections. The number of supported connections that a component has is unlimited; a beam or column may support any number of other components through supported connections. Because each supporting connection is special, the forms `Beam` and `Column` have individual slots for each supporting connection. In the form `Column` the single supporting connection is stored in the slot `Base-connection`. In the form `Beam` the supporting

connections are stored in the slots `Connection1` and `Connection2`. Supported connections are stored together in a single multiple-value slot. Both forms, `Beam` and `Column`, store values for their supported connections in a slot called `Connections`.

The location of a component is determined from the locations of its supporting connections, and the location of a connection is computed relative to its supporting component. Since the parameters of the forms which define relative locations vary depending on the component type—for example, the location relative to a beam is determined from the `End1` slot of the `Beam` form, and the location relative to a column uses the `Elevation` slot of the `Column` form—the appropriate parameters of the relative location, in the form `Connect`, are selected by the method `Select-rel-loc`.

The knowledge base forms include methods for computing gravity loads. These loads are transferred from the supported component to the supporting component through the shared connection. External loads are applied only to beams.

The global form `Connection` has no method for load transfer. The load transfer method is defined in the local specializations of `Connect` in the forms `Beam` and `Column`. The load transferred by the base connection of a column is the sum of the loads of its supported connections. The method `get-load` is used in the local form `Beam` to serve this purpose. Unlike the form `Column`, the form `Beam` includes parameters for the application of external loads. Both point loads and uniformly-distributed loads are accepted. The total load on the beam is computed by the method `get-total-load`. This load is distributed between the two end connections.

The current implementation of connectivity allows any component to connect to any other. Thus, a beam may connect to either a column or another beam, and a column may connect to another column or a beam. Because a column, as defined, must connect to another component, a new component, represented by the form `Foundation`, is defined to support the main building columns. A foundation is a non-physical component that accepts load without transferring it. The form `Foundation` computes its load as the sum of the loads of its supported connections. In effect, the `Foundation` form presently represents a simple reaction for gravity loads.

Another form that is introduced is `Structure`. The purpose of this form is to combine all components into one entity, a `structure`. Thus, the form `Structure` is an assembly of `Column-line`, `Elevation`, `Foundation`, `Column`, and `Beam`.

7.2 Description

To illustrate the creation and use of a description, a two-story structure, whose floor and roof framing plans are shown in Figure 6, 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.

The following dialogue is from a session on the system in which the user is creating a description of the building shown in Figure 6. 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 7 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 floor elevations of the base, `Floor1`, by providing the floor identifier, floor definition, and the elevation of the floor. For example,

```
[6] (create-instances 'elevation)
Instance id: floor1-e1
Instance defn (string): "Elevation of Floor1"
Y: 6250
```

Next, the column-lines are defined. This involves entering the column-line identifier, `L1`, 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,

```
[7] (create-instance 'column-line)
Instance id: L1
Instance defn (string): "Column Line L1"
POINT: VECTOR-ZX
X: 0
Z: 0
ROTATION: 0
```


Mem. T.O.S. 10000

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 L1 and L2.

[8] create-instance 'foundation'

```
Instance id: F1
Instance defn (string): "Foundation at base of column C1"
ELEVATION: 1250
LOCATION-ZX: column-line-intersection
COLUMN-LINE-1: L1
COLUMN-LINE-2: L2
```

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.

[9] create-instance 'column'

```
Instance id: C1
Instance defn (string): "COLUMN C1"
COMPONENT: 1
**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: 1000-e1
CROSS-SECTION: n11
```

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.

[13] create-instance 'beam'

```
Instance id: B1
Instance defn (string): "Beam B1"
COMPONENT: C1
ELEVATION: floor1-e1
COMPONENT: C2
ELEVATION: floor1-e1
POINT-LOADS: n11
DISTRIBUTED-LOADS: 16-55
L1: 0
L2: 2000
DIST-LOAD: 16.67
DISTRIBUTED-LOADS: 14-55
L1: 4000
L2: 7000
DIST-LOAD: 17.5
DISTRIBUTED-LOADS: n11
CROSS-SECTION: n11
MATERIAL: n11
```

Finally, all these components are combined in an instance of the form `Structure` by entering all the columns, foundation elements, beams, and columns that make up the structure. Specifying a value ends the entry of that type of information and the system proceeds on to additional prior examples.

[49] (`create 'bidg-structure'`)

Instance of: `"Assembled structure"`

COLUMN-LINE

COLUMN-LINE

FOUNDATION

FOUNDATION

COLUMNS:

COLUMNS:

BEAMS: 8

BEAMS: 8

The above data is incomplete because the cross-sections of the structure components have not been input. The default values of some important properties can be computed. Since the computational load does not depend on the properties of the component cross-sections, the structure load is computed as hard values.

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

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

[29] (`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.

8 Conclusion

This paper described a specific implementation based on the conceptual frame 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 frame 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.

9 Acknowledgement

Thanks are due to Don Duchesne of Adjeleian Allen Rubell 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

- [ACM85] Special issue on knowledge representation. *Communications of the ACM*, 28:9, 1985.
- [Ahmed91] Ahmed, S., A. Wong, D. Sritam, and R.D. Logcher. "Object-oriented database management systems for engineering: a comparison." Technical report, Intelligent Engineering Systems Laboratory, Department of Civil Engineering, Texas Institute, Cambridge, MA, July 1991.
- [Barbucaanu84] Barbucaanu, M. "Object-centered representation and reasoning: an application to computer-aided design." *SIGART Newsletter*, 87:33-39, 1984.
- [Barbucaanu85] Barbucaanu, M. "An object-centered framework for expert systems in computer-aided design." *Knowledge Engineering in Computer-Aided Design*. J.S. Gero, ed., Amsterdam: North-Holland, 1985, pp. 223-253.
- [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.
- [Fikes85] Fikes, R., and T. Kehler. "The role of frame-based representation in reasoning." *Communications of the ACM*, 28:9:904-920, 1985.
- [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.
- [Hartquist83] Hartquist, E.E., and H.A. Marisa. *PADL-2 User's Manual. Early release*. Production Automation Project, College of Engineering and Applied Science, University of Rochester, Rochester, New York, May 1983.

- [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.

- [Minsky75] Minsky, M. "A framework for representing knowledge." *Mind Design*, J. Haugeland, ed., Cambridge, MA: MIT Press, 1981, pp. 95-128. Excerpted from Memo 306, Artificial Intelligence Laboratory, MIT. Also excerpted in *Psychology of Computer Vision*, P. Winston, ed., New York: McGraw-Hill, 1975.

- [Oxford83] *The Oxford Paperback Dictionary*, London: Oxford University Press, 1983.

- [Rasdorf92] Rasdorf, W.J., and O.Y. Abudayeh. "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.

- [Steff86a] Steff, M., and D.G. Bobrow. "Object-oriented programming: themes and variations." *The AI Magazine*, 6:4:40-62, 1986.

- [Steff86b] Steff, M., D.G. Bobrow, and K.M. Kahn. "Integrating access-oriented programming into a multiparadigm environment." *IEEE Software*, 3:1:10-18, 1986.

- [Winston77] Winston, P.H. *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1977.

Contents

1	Introduction	1	7	Example	19
2	Overview of the Frames Technique	3	7.1	Knowledge Base	19
3	Scope	4	7.2	Description	22
4	The Description System	5	8	Conclusion	28
4.1	The Shell	5	9	Acknowledgement	29
4.2	The Knowledge Base	7			
4.2.1	Forms	7			
4.2.2	Comparison of Forms to Frames	8			
4.3	The Description	10			
5	Features of the Description System	11			
5.1	Data-Directed Computations	11			
5.2	Data Integrity	12			
5.2.1	Dependency Links	12			
5.2.2	Change Propagation	13			
5.3	Default Values	14			
5.3.1	Soft Defaults	15			
5.3.2	Integrity Maintenance	16			
6	User Interface	17			
6.1	Forms and Instances	17			
6.2	Menus	18			