AN INFORMATION MODEL FOR THE PRELIMINARY DESIGN OF BUILDINGS

Rivard H., Gomez N. and Fenves S. J.

ABSTRACT: This paper presents the current status of the information model of SEED-Config, one of the modules of the SEED (Software Environment to support the Early phases in building Design) prototype being developed at Carnegie Mellon University and the University of Adelaide. The goal of this information model is twofold: to store the design data as it is generated during the conceptual design and to support case-based reasoning. The main constructs of this information model are building entities, containment and domain specific relationships, technologies, components, groups, and classifiers. Design knowledge is represented through technology hierarchies. This information model is being implemented in SPROUT, an information language. The paper concludes with a discussion on the use of this information model in support of case-based reasoning.

KEYWORDS: Product modeling, Conceptual Design, Structural Design, Case-Based Reasoning

1. INTRODUCTION

SEED (Software Environment to support the Early phases in building Design) is a multidisciplinary project that involves the Engineering Design Research Center, the Department of Architecture, and the Department of Civil and Environmental Engineering at Carnegie Mellon University and the Department of Architecture at the University of Adelaide. SEED consists of three main modules: SEED-Pro, which supports the generation of architectural programs; SEED-Layout, which supports the generation of schematic layouts; and SEED-Config, which supports the generation of 3-dimensional configurations of spatial and physical building components (Flemming and Woodbury, 1995). This paper presents the current status of the information model for the SEED-Config module, updating the previous presentation in (Rivard et al., 1995).

The information model presented here addresses the early design stages and supports design evolution and the rapid generation and evaluation of alternative solutions to a design problem. This contrast with current efforts for developing Standard for the Exchange of Product Model Data (STEP) product models, which typically address later stages of design and do not explicitly support design evolution (Wix and Bloomfield, 1995).

2. THE INFORMATION MODEL

This section describes the main concepts of the information model of SEED-Config. First, the building entities and their categories of information are described. The second sub-section provides a description of the two types of relationships supported in the model. This is followed by a description of how building entity attributes are aggregated into components. Then, the technology hierarchy, which represents design knowledge in SEED-Config, is briefly described. The mechanism for grouping entities is discussed next. Finally, the classification of building entities through the use of classifiers is described.

The goal of this information model is twofold: to store the design data as it is generated during conceptual design and to support case-based reasoning. Since the representation is identical for the two, no translation is required to store the design data into a case. This simplifies the implementation of the premise stated in (Flemming, 1994) that cases should be accumulated as a side-effect of generative design activities.

2.1 Building Entities

The building is represented as an assembly of entities with relationships among them. Each entity represents a concept meaningful to design participants such as a beam, a room or a structural frame. An entity can be a system, a sub-system, a component, a part, a feature of a part, a space or a joint (Gielingh, 1988). An entity includes the following categories of information: functional unit components; design unit components; evaluation unit components; spatial representations; relationships; technologies; and classifiers. Figure 1 shows, using the OMT notation (Rumbaugh et al., 1991), the relationships between the building entity and the other constructs to be discussed in the information model.

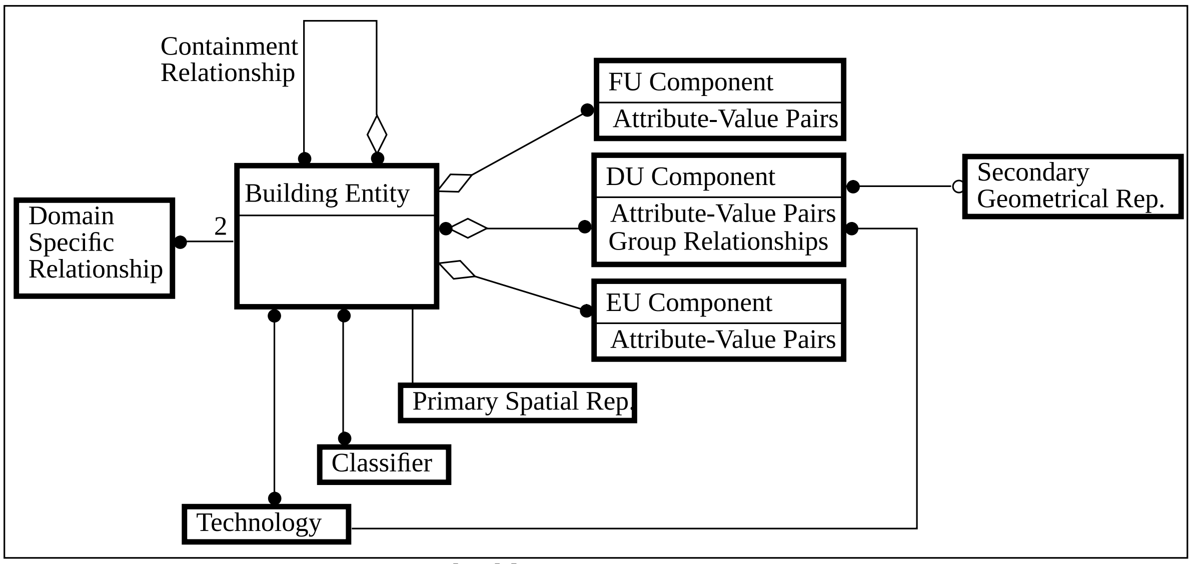

FIG. 1: A building entity using OMT notation.

Design requirements, design descriptions, and behavior evaluations of an entity can be represented as attribute-value pairs where values may be an atomic type, a matrix, a derived value (which may depend on other attribute-value pairs), a diagram, and so on. In this model, a value cannot be a geometrical description, a relationship with another entity nor a reference to a technology. The attribute-value pairs characterizing an entity are organized at two hierarchical levels: the unit level and the component level (described in sub-section 2.3). At the top level, data are grouped into three subsets: the functional aspect, the design aspect and the behavior aspect. The functional aspect includes the intended purposes, requirements and constraints on the entity; this aspect is called the functional unit (FU). These requirements have to be satisfied in order to realize the intended purpose. The functional unit can be seen as a design-problem statement (Gielingh, 1988). The design aspect includes all the physical and spatial characteristics that define the actual design of the entity; this is called the design unit (DU). The design unit can be seen as a solution alternative to the design-problem. The behavior aspect includes the response to stimulations associated with different design conditions, and is called the evaluation unit (EU). The evaluation unit gives access to the behavior computed by external applications without the re-computation cost every time this information is accessed. It can also be used to record the designer's comments after implementation. Examples of data recorded in the evaluation unit are costs, stresses, heat flow, condensation rate, and in-service performance.

The geometrical description of an entity is classified in two categories. The primary spatial representation is its high-level geometric description which is used primarily for reasoning about its topological relationships with other entities, while secondary representations of an entity are its domain-specific geometric representations. Each entity has only one primary spatial representation, but it may have several secondary representations which are part of the design unit. This representation scheme relies on the premise that topological relationships of physical entities are invariant with respect to their domain-specific secondary representations (Zamanian, 1992). A non-manifold boundary representation scheme is used, since topological relationships can be represented without being affected by the various dimensionalities used in representing the geometric entities.

2.2 Relationships

Relationships among entities are classified into two categories: containment relationships (also known as aggregation relationships) which capture the link between an entity and its parts, and domain-specific relationships which contain other relationships of interest. The containment relationship supports the hierarchical decomposition of the design problem. Complex artifacts with their parts can be modeled as entities linked by containment relationships. The result of the hierarchical decomposition is a tree of entities. The designer can look at the design at any level of abstraction simply by going to the corresponding depth in the tree. The complex object is treated as a unit in many operations, although physically it is made of several parts. Other essential relationships are stored in the second category of relationships, called domain-specific, and are further described below. Spatial relationships, such as next to, above, spatially contained in, and adjacent are not stored explicitly in this information model since they can be obtained directly from the geometric modeler.

A building entity may have several distinct domain-specific relationships (DSR) with other entities. Examples are **gravity-supports**, **lateral-supports**, **controls**, and so on. Even though a DSR refines the design of a building entity, like a DU component, it is stored at the building entity level for ease of access. Each DSR is implemented as an object which specifies the roles of the two building entities that it relates. DSR's are created and assigned to the two related entities by the technology that identified the relationship. For instance, a **gravity-supports** DSR, which is used to define a load path among two structural entities, would have **supports** and **supported-by** as the roles fulfilled by the two related entities. Figure 2 shows how two building entities are related through a DSR. The different types of DSR are implemented as different specializations of the generic DSR class.

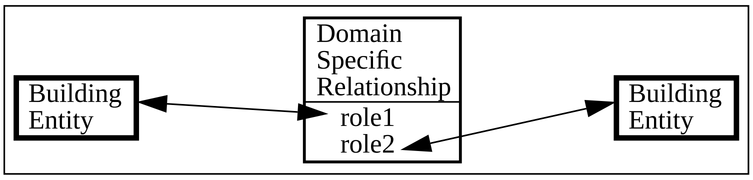

FIG. 2. Representation of a domain-specific relationship.

2.3 Components

At the second level of data aggregation, the attribute-value pairs of an entity are combined into small cohesive subsets, each of which is called a component. Component is preferred to the term “primitive” which was introduced in (Rivard et al., 1995) based on (Howard et al., 1992). This approach supports multiple discipline-related views, schema evolution and data integration. Another advantage of encapsulating data within component objects is that the data stored can be of multimedia type, since the component has the appropriate methods to display, edit and input its data. Thus, a component could contain images, sounds, texts and even video.

The definitions of the FU, DU, and EU components are inherited from a common superclass called Component. These generic classes are shown in Figure 3. Every component stores the name of the designer who is responsible for its creation, refers to the building entity to which it belongs, and has a built-in help mechanism that provides a description of the data stored in it. A component can have both attributes and methods (or procedures). These component methods are used to compute values based on other attributes or components and provide support for dependencies among data (e.g., the area of a rectangle can be obtained by multiplying its length and width). The specialized DU component class stores, in addition, a reference to the technology node that was used to instantiate it, a status which could take one of three values: candidate (alternative is not explored yet); explored (alternative has been explored but not selected); and committed (alternative represents the current design), and references to the group the DU component is a member of and to the building entities that own the DU component (see sub-section 2.5). The actual components stored in a building entity are instances of classes that specialize the FU, DU, or EU generic component classes.

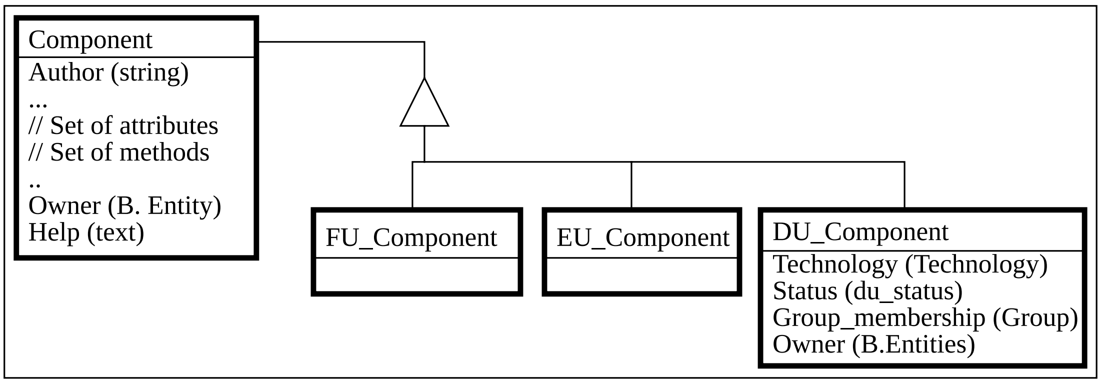

FIG. 3. The generic component classes.

The information representing a building entity, encapsulated within components, is accessed through a search mechanism. First, the requester (which may be a user or a computer application) must specify one of the categories where the search is to be made: functional unit, design unit or evaluation unit. Second, the requester must provide the name of the component class that is of interest. There can be only one instance of a component class stored in a building entity. Hence, the name of the component class uniquely identifies one subset of attributes. For example, a requester interested in checking the concrete characteristics of a structural element needs to provide the following request to the building entity: design unit (concrete characteristics). The system then looks among the components stored in the design unit and returns the one with the matching name if it is found.

2.4 Technologies

The set of technologies is SEED-Config's encapsulation of design knowledge. The knowledge required to implement a design utilizing a specific constructed system or component (e.g., a rigid frame configuration or a reinforced concrete slab) is stored in a technology node. Thus, the knowledge base is modularized into nodes which are then organized in a hierarchy ranging from the most abstract concept (e.g., braced frame, rigid frame) to the most specific (e.g., a reinforced concrete beam). A technology node also stores the range of applicability of the constructed system. Technologies advance the design of a building entity by creating DU components, classifiers, and relationships according to the domain knowledge, and appending them to the building entity.

Since many design tasks require decomposition into subtasks for ease in problem conceptualization and solution, technology nodes are employed to aid in decomposing more abstract entities into more manageable and more specific entities (e.g., a frame may be decomposed into beams, columns, and connections, or a steel deck-on-joists slab can be decomposed into its joist and slab components). Technology nodes which perform this kind of elaboration create new, less abstract building entities and link them to the original entity through the containment relationship. These elaboration technologies are rare in the hierarchy, as compared to refinement technologies which perform the bulk of the design activities by expanding the descriptions of the building entities.

Also, if the problem specifications change, technologies may be employed to modify an existing design by pointing out which previously used technology nodes, if any, are no longer applicable and which alternate technology nodes may be utilized. Figure 4 presents an example, illustrating design evolution, where the design context is changed and as a consequence, previous choices become inapplicable. In the figure, solid nodes and dashed nodes represent

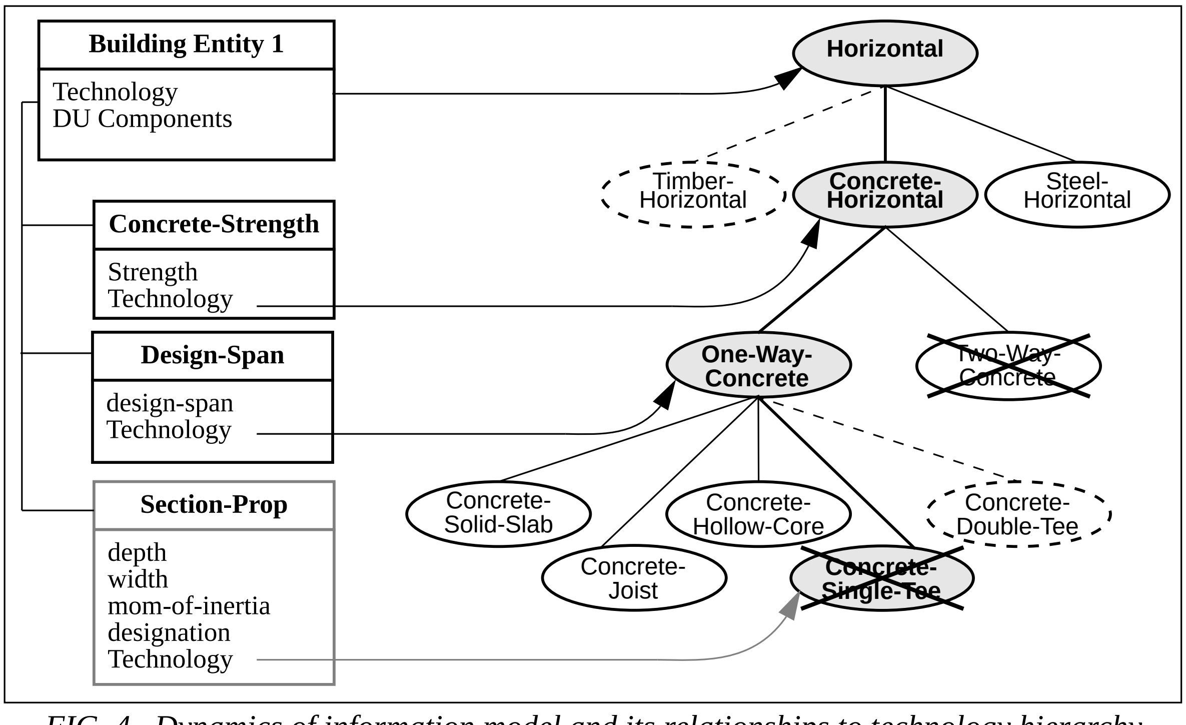

FIG. 4. Dynamics of information model and its relationships to technology hierarchy.

technologies that were respectively applicable and inapplicable before the change, and shaded nodes show the selected ones, resulting in the creation of the DU components shown on the left. After the change, the subset of the technology hierarchy associated with the current building entity is reassessed in the modified design context; the crossed-out nodes symbolize previously applicable technologies that can no longer be used to solve the problem. The designer only needs to re-select an alternative applicable technology node to complete the design (e.g., Concrete-Solid-Slab). This method of redesign is specially useful in case adaptation, an important part of the case-based reasoning capabilities of SEED-Config.

2.5 Grouping Entities

When several building elements possess some similarity, designers typically consider only the most constraining one and apply that design to all of the elements. This is typical because it facilitates the design, construction, and management of these entities. In SEED-Config, a set of entities can be grouped together and designed simultaneously. Three different types of groups have been identified: **same** means that all the entities are assigned the same design components according to the most constraining entity of the group; **similar** means that all the entities are designed with the same technology but may have design components with different attribute values; and **identical** means that all the entities in the group have the same dimensions and are subject to the same conditions and hence can be designed just by designing one of the entities. As for **same**, the entities grouped by **identical** are assigned the same design unit components. While the two groups **same** and **similar** are specified by the user, the group **identical** is created by default by elaborating technologies.

The grouping is done at the level of DU components because one building entity can be in several different groups based on different DU components. For instance, several slabs could be grouped in such a way that they have different depths, the same material, and similar reinforcements. Each characteristic (i.e. depths, material and reinforcement) corresponds to different DU components.

The grouping concept is illustrated with an example of the use of **same** group. In this example, a structural engineer decides to assign the same concrete characteristics to a group of entities. A number of entities are assembled in a **same** group, which is then passed as an argument to the desired refining technology. Once the technology has verified its applicability with each entity within the group it creates one DU component with the concrete specification and assigns it to all the building entities of the group. Having only one DU component for a group of entities ensures that any changes will be applied to all entities (i.e. there is no redundancy). This example is illustrated in Figure 5 below. For a group of **similar** entities, the technology would add a new DU component to each building entity.

Another example illustrates the use of **identical** grouping. When elaborating a structural slab into intermediate or "filler" beams and slab segments, the technology creates a number of beam-entities and deck-entities. All these entities are associated with the slab entity through the containment relationship. Each distinct sub-entity is needed because each one has its own distinct primary spatial representation. To simplify design, all the beam-entities are automatically grouped into an **identical** group and thus consistency between the beams is ensured. When a technology refines the beam-entities, it assigns the same DU components to all of the entities in the group. The same process applies for the deck-entities. Figure 6 below illustrates the creation and design of the beam entities.

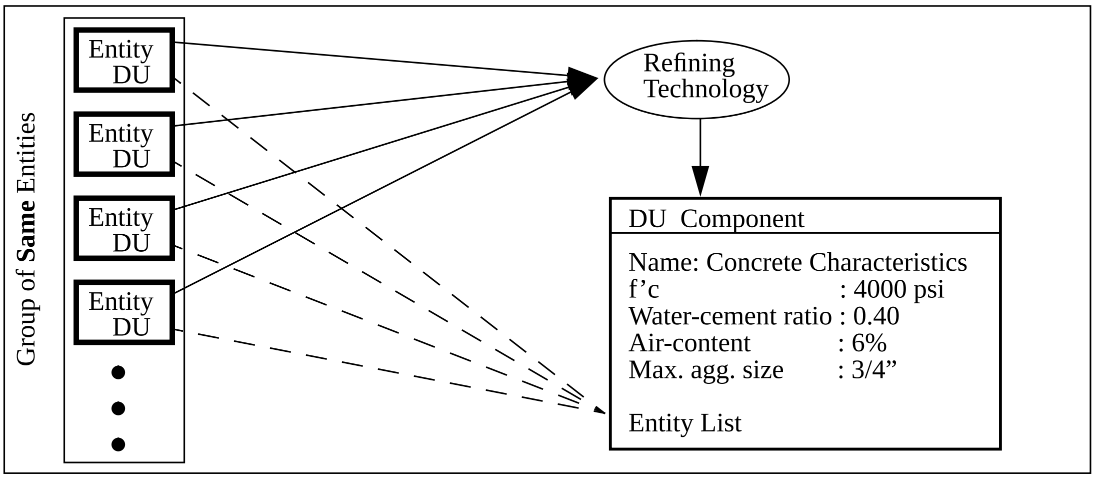

FIG. 5. Representation of a group of **same** *entities.*

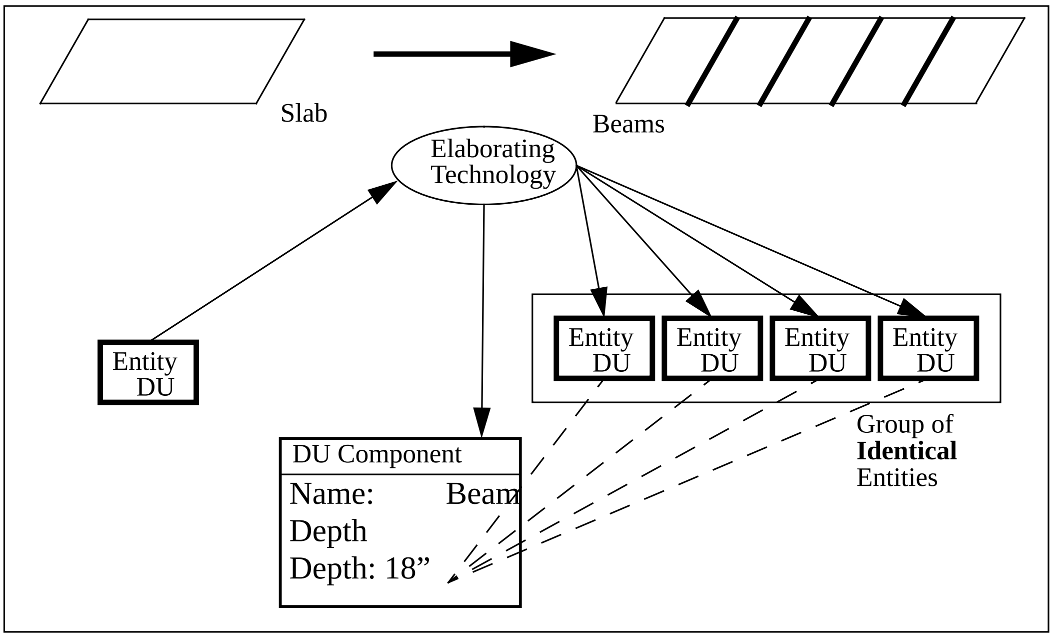

FIG. 6. Representation of a group of **identical** *entities.*

2.6 Classifiers

A classifier is a label assigned to a building entity. Classification is the process of systematically arranging entities into conceptual groups and is an important aspect of problem-solving tasks. The classification process assigns to a particular entity the name of a class to which it belongs. In design, the designer unconsciously assigns labels to the building entity description being generated. This label assignment is a classification.

Classifiers are required to classify the generic building entities as they are being refined during the design process. They categorize entities in domains and sub-domains. Classifiers are also used as indexes for querying the database and for retrieving cases. They provide access to building entities independently from the design sequence that generated them. A given building entity may have several classifiers assigned to it, each of which offers a different access to retrieve it. Hence, a search can be based on a unique classifier or on the conjunction of a set of classifiers.

Classifiers are assigned automatically to the building entity by the technologies selected by the designer. A classifier is used whenever the value of an attribute is a symbolic value selected from a predefined set. An example of a predefined set of symbolic values is the type of structural material: steel, concrete, timber and so on.

The classifiers are arranged in a classification hierarchy. Classifiers are more specific as one travels down and more general or abstract as one travels up the hierarchy. Classification hierarchies are in fact semantic networks, which provide an efficient mechanism for supporting reasoning about the generalization or specialization of concepts. This is essential for widening the space of applicable cases when a query results in a limited number of matches. Say, for instance, that the retrieval of a structure to help design a swimming pool facility has been unsuccessful because no similar cases exist in the case base. The query can be generalized, through the semantic network depicted in Figure 7, to sport facilities or even to halls in order to find, possibly, the structural design of an auditorium that could be used as a starting point.

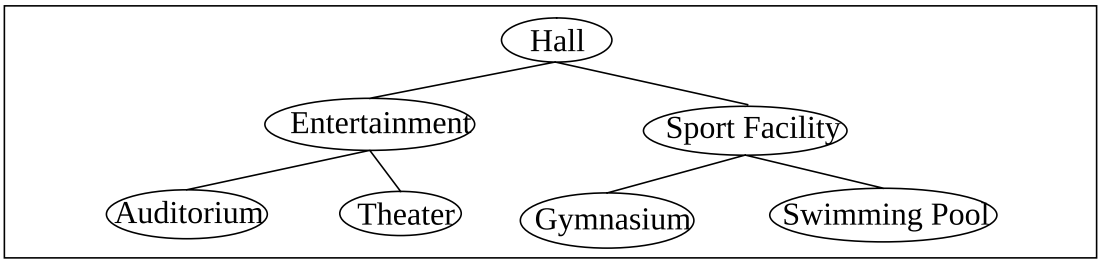

FIG. 7 A classification hierarchy.

The KL-ONE family of knowledge representation (Brachman and Schmolze, 1985) is very efficient at representing and reasoning about classifications. We are planning to use one of these systems, CLASSIC, to implement the case retrieval system and the database query facility of SEED-Config. CLASSIC will function as a knowledge server to build, manage, infer about, and query the classification hierarchy.

CLASSIC is a frame-based knowledge representation system which differentiates between terminological and assertional aspects of knowledge representation, and which focuses on the key inferences of subsumption and classification (Brachman et al., 1991). The terminological knowledge aspect describes classes of individuals through the use of concepts and their classifications, while the assertion knowledge aspect records constraints or facts that apply to a particular domain (Mac Gregor, 1991). A subsumption inference is the process of recognizing a concept to be part of a larger encompassing concept. CLASSIC supports complex information retrieval, handles evolving schemas, manipulates partial knowledge or incrementally evolving descriptions, automatically detects inconsistent knowledge (with a truth-maintenance facility), and enforces constraints (implemented as "trigger"-like rules) on collections of facts through strict inheritance.

3.0 KNOWLEDGE REPRESENTATION THROUGH TECHNOLOGIES

To be of greater benefit to the design process and to model design knowledge efficiently, technologies must do more than model specific real-world construction solutions. In order to allow the design to proceed in meaningful steps, the technology nodes must be arranged so that high-level decisions can be made without over-committing to a design methodology (e.g., a material such as reinforced concrete can be chosen without having to select a structural configuration such as waffle slab). Technologies must be able to query information from the design context represented by the collection of building entities (e.g., calculate loadings by following load paths, getting topological information, etc.).

In addition, technologies must be able to perform in many design situations. They must be usable in SEED-Config's three generation modes: **manual**, in which the designer attempts to apply a selected technology node; **interactive**, in which the system guides the designer by displaying all applicable expansions at the next level from the current state; and **automatic**, in which the system expands every possible alternative to a target level. In all three modes, the designer must explicitly commit to one alternative. As described in section 2.5, groups of building entities may be designed simultaneously and technologies must handle that contingency as well.

A technology node maintains relationships only with other technology nodes, but it can interact with other entities when the latter are sent as parameters or are created by the technology node. Technologies can be used in the three generation modes, in simultaneous design, and in redesign. Which behaviors a technology takes on depends solely on what accesses the technology (e.g., a group of building entities or a single building entity). In general, a technology node is defined by three components: how it relates to other technology nodes, what constraints it must meet to be applicable, and what its output is. The first defines prerequisites, the second defines applicability, and the last defines the design action. These make up the interface of a technology node as used by the other entities in SEED-Config.

3.1 Prerequisites

How a technology node relates to other nodes describes where the node fits in the technology hierarchy. Thus, the hierarchy is maintained by the inter-relationships of the nodes themselves, not by some higher construct. This follows naturally from object-oriented methodology and makes the hierarchy more maintainable. When applied to a building entity, a technology node must make sure that any prerequisite nodes have already been applied to that entity. The prerequisites may be described in any boolean combination (e.g., a concrete solid slab technology requires that the material technology and an "action" technology (one-way or two-way) be previously selected). This prerequisite check may be easily performed by querying the building entity's DU components, which always maintain relationships to the technology nodes.

The inverse relationship of "prerequisite" is "possible-successor". These latter relationships, which are automatically maintained by the implementation environment, SPROUT, are used to guide the design after the current node is applied. The successor nodes can be used as the next level of expansion to be explored. In automatic mode, all applicable immediate successors are generated and explored, whereas in interactive mode, all applicable successor nodes are presented to the designer.

This approach to modeling the hierarchy offers many benefits over our previous strict tree representation. The biggest advantage is that the user is no longer forced into making design decisions in a rigid order. The hierarchy approach also increases modularity of the knowledge representation. This will make it easier to create and append user-defined technologies or to reorganize the hierarchy to better match certain project contexts. The arrangement of the hierarchy is expected to mainly be a function of data modeling, but correlations to the domain knowledge must be maintained.

3.2 Applicability

Once the prerequisites of a technology node are met, its applicability must then be tested. As mentioned earlier, constructed systems generally have constraints on their applicability (e.g., a dome is applicable for large indoor spaces but not for single-family housing, whereas the opposite can be said of a timber pitched roof). Constraints are associated with technology nodes based both on feasibility (e.g., a timber beam cannot meet high fire-resistance requirements) and on economy of design (e.g., although a concrete solid slab can be made deep enough to resist a load over a particularly large span, a concrete joist slab is more efficient). Engineering expertise and heuristics (e.g., see (Schodek, 1980)) are used to determine appropriate constraints for the more abstract technologies. For more specific technology nodes, constraints are obtained from product catalogs and/or design standards.

Constraints are established in different ways. They may take the form of geometric limitations such as a suitable span range or a required aspect ratio. The technology must retrieve these values from the primary spatial representation or from DU components linked to the current building entity. Constraints may also take the form of usability requirements such as loadings, minimum deflections, and voids required. The technology must obtain these values from functional unit components associated with the current building entity. In addition, since some technologies must be used in combination with other technologies (e.g., a pre-cast horizontal system should not be supported by an unbraced frame, whereas a cast-in-place system can be (Schodek, 1980, p. 484)), the technology may also need to check if related building entities are implemented using compatible technologies. This necessitates that related entities be obtained from the current building entity and checked for compliance.

3.3 Action

Action specifies what a technology must do to expand the current building entity once both the prerequisites and the applicability constraints are met. Mostly, technologies expand the building entity by refining its design, and are thus termed refinement technologies. This refinement process may include the addition of DU components, classifiers, or domain-specific relationships. For example, a concrete one-way slab is refined into a concrete single tee by the addition of a DU component describing cross-section properties which are calculated based on span and loading, and the addition of a classifier indicating the use of pre-fabricated elements (see Figure 3). Relationships are appended when necessary. The supports/supported-by relationship, for example, only makes sense when the structural action (e.g., one-way or two-way) of a horizontal 2-dimensional entity is selected.

For elaboration technologies, the process specifies what contained building entities need to be created, what relationships, if any, should exist between them, and which technology nodes should be associated with the new entities. The methods to carry out the elaboration are also included in this section of the technology node. The existence of these methods is what identifies an elaboration technology, whereas the existence of DU component and classifier appending methods identifies a refinement technology.

4.0 SPROUT SPECIFICATION

SPROUT (SEED representation of Processes, Rules, and Objects Utilizing Technologies) is an information modeling language that supports the specification of information models, and particularly building product models (Snyder et al., 1995). SPROUT's goals are to administer the persistent storage of objects, to support case-based design, to foster communication between different SEED modules, to manage versions, and to integrate external applications. Complex information models can be specified in this neutral schema definition language. The resulting schemas are used as an interface between the module (e.g., SEED-Config) and the object-oriented database, other modules and external applications. SPROUT can provide objects in the representation required by the retrieving module even if the object was originally generated by a module using a different internal representation. The foundation for all these capabilities are the schemas, or SPROUT specifications, from which SPROUT can automatically generate the required computer programs and database schemas. SPROUT can be seen as the "software glue" that integrates the various SEED modules along with the various support software that are being used in implementing SEED (i.e., the database system UniSQL, the interface builder ET++, the geometric modeler ACIS, and the knowledge representation system CLASSIC).

The SPROUT specification of a model serves as the basis for an application to communicate with the database, other modules and with support software. A portion of the SPROUT specification for the information model presented in this paper is shown in Figure 8. This example illustrates the description of three classes: the Component superclass; the DU_Component class; and an example of a DU component class. The expressive power of SPROUT cannot be presented here due to space consideration. However, the example gives a sense of what the SPROUT syntax looks like. Items in bold are SPROUT reserved words. A class is a list of slots which can be either a value or a relationship. The definition of a value (or an attribute) consists in the name of the slot followed by its domain or class. Domains such as real, string and text are provided by SPROUT while other domains such as du_status are defined specifically for the model. The definition of a relationship consists in the name of the slot followed by the class of objects referred, within parentheses, and the type of relationship.

```
class Component is
        value author string; // Name of designer who created it.
        value help text; // Text describing the component.
        end

class DU_Component superclass Component is
        value tech Technology;
        value status du_status;
        relationship entities (BuildingEntity) many_to_many;
        relationship group_membership (Group) many_to_many;
        end

class Concrete_Characteristics superclass DU_Component is
        value fc real;   // Compressive strength of concrete.
        value water_cement_ratio real;
        value air_content real;
        value max_agg_size real;
        end
```

FIG. 8 Example of a SPROUT specification.

Versions are ubiquitous in design. In SEED, version management is handled by SPROUT through a time travel approach. In this approach, each database instance is identified by a twotuple containing a logical object identifier (LOID) and a time stamp. The LOID is persistently stored in the database and keeps track of its instances which have different time stamps. Only one of these instances is identified as the current one. This approach supports the following operations: **overwrite** which updates the time stamp to the current time and, if there has been changes, records the changes onto the current instance hence deleting the old instance; **anchor** which creates a new instance with the same LOID but with a new time stamp hence creating a new version; **delete** which permanently removes the instance from the database; and **copy** which makes a new instance of an object with a different LOID. Figure 9 illustrates the operations and an example of version management where each number represents a version and each arrow corresponds to one of the operations. These operations can be propagated over relationships depending on how they have been defined by the user hence providing a great control over configuration management.

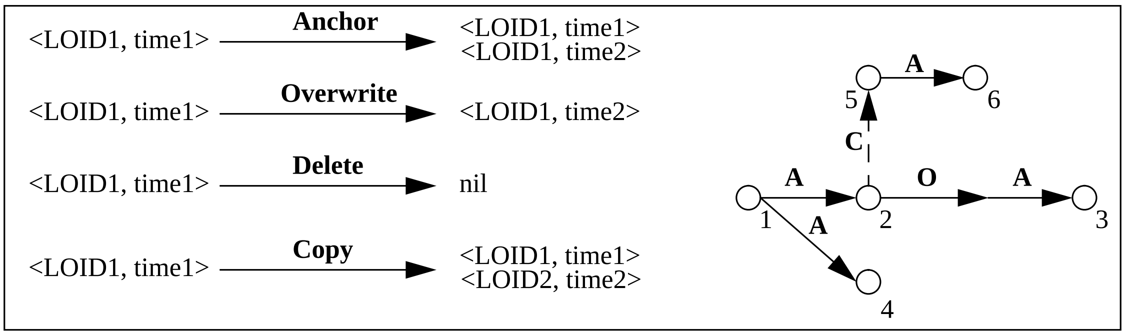

FIG. 9. The operations and an example of version management.

Alternatives, which represent different solutions for a given problem, are not directly managed by SPROUT. Each SEED module must implement its own alternative model since each has a different notion of what is an alternative. In SEED-Config, the designer is encouraged to explore any or all of the alternative technologies applicable at any stage of design expansion; however, only one of these technologies may be committed (i.e., instantiated) in a particular version of the emerging design database. To initiate a new alternative, the state of the building entities just prior to the alternative creation is reconstructed (if necessary, by deleting later DU components, relationships, and contained building entities) and anchored. Further expansion starts from the newly created building entity instance. Controls will be implemented to test whether an alternative is complete in the sense that all functional requirements are satisfied.

5.0 CASE-BASED REASONING SUPPORT

Since creating a new design requires prior experience, or exposure to someone else’s design experience, a computer tool that would store prior design experiences and make them available to designers could be of great help for seasoned as well as for novice designers. A design system incorporating case-based reasoning represents such a computer tool. It could help designers to remember previous and appropriate cases which could be used as sources on the basis of which a more relevant solution to the current problem can be developed. Designers use previous designs because they do not like to waste time solving problems that they have already solved before and because the concepts have been tested and proven effective in the past. Case-based reasoning is an attempt to implement this natural design process in computers (Kolodner, 1993). This section describes how the information model presented here provides support for case-based reasoning.

The case representation in SEED-Config supports the hierarchical decomposition of design cases. Every level of a containment hierarchy is represented in the same way as a building entity. Hence, the parts of a case are as simple to access as the actual global case. Because each part is stored as a whole, it is indexed independently. It can thus be accessed independent of the global case. Furthermore, the grouping of attribute-value pairs in components supports the retrieval of cases (or entities) with multiple functions (or views). Multifunction entities are frequent in building design and must be supported by a case-based reasoning system.

The division of a case (i.e., building entity) data into functional unit, design unit and evaluation unit allows retrieval to be based not just on the problem specification, but also on the solution and on the outcomes. The performance of a design can be tested by comparing the behavior stored in the evaluation unit with the requirements set forth in the functional unit. Dividing the case's data into these three subsets provide greater flexibility, guides the examination of the content of a case, and classifies the attributes according to their role within the design process (Maher et al., 1995).

Classifiers provide a powerful means for retrieving cases. Classifiers are used as a deep indexing mechanism that allows generalization or specialization of queries. Also, the fact that a component refers back to the entities it belongs to allows a case search to be done from bottom-up. Hence, the name of the component classes can also be used as an index to limit the search to a particular type of component. It is also possible to limit the search of a case to a given technology or to exclude a given technology from the search. Once a set of cases (or building entities) are retrieved based on one of these three mechanisms, the case that best matches the current situation can be found through a matching and ranking process.

Both the solution and the reasoning steps are stored in a case. The reference to the technology nodes actually refers to the knowledge used in designing the case or entity. Technologies can be seen as solution operators. They can be re-used for rapid adaptation or for redesign of a case. The set of technologies used in designing an entity also provides a limited way of describing how its solution evolved.

6.0 CONCLUSIONS

The information model presented is specifically designed to serve two objectives: to assist designers to rapidly generate alternative conceptual structural configurations, and to serve as a case base of previous designs so that relevant cases can be recalled and adapted as elements of the solution of new design problems. The overall goal is to assist the structural engineer in the exploration of the total building design in collaboration with the owner, architect and other design professionals participating in the conceptual design phase by evaluating structural consequences of architectural decisions and by suggesting architectural configurations that result in effective structural schemes. For recurring building types (e.g., hospitals, schools, etc.) a case base of previous designs can greatly enhance the engineer's ability to develop appropriate and proven configurations or components.

Case-based reasoning and the encapsulation of generative knowledge into technologies serve the further purpose of providing customization potential to individual user organizations, so that the designs produced can reflect each organization's design style and accumulated expertise.

References

Brachman, R. J. and J. G. Schmolze (1985). “An overview of the KL-ONE knowledge representation system.” Cognitive Science, Vol. 9, No. 2, pp. 171-216.

Brachman, R. J., D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida (1991). “Living with CLASSIC: When and How to Use a KL-ONE-Like Language”, Principles of Semantic Networks, John F. Sowa Editor, Morgan Kaufmann Pub. Inc., San Mateo, pp. 401-456.

Flemming, Ulrich (1994). “Case-Based Design in the SEED System.” *Knowledge-Based Computer-Aided Architectural Design*, G. Carrara and Y. Kalay (editors), Elsevier, New-York, pp. 69-91.

Flemming, Ulrich, and Robert Woodbury (1995). “Software Environment to Support Early Phases in Building Design (SEED): Overview.” Journal of Architectural Engineering, ASCE, Vol.1, No. 4, pp. 147-152.

Gielingh, W. (1988). “General AEC Reference Model.” ISO TC 184/SC4/WG1 doc 3.2.2.1, TNO Report BI-88-150.

Howard, H. C., Jamal A. Abdalla, and D. H. Douglas Phan (1992). “Primitive-Composite Approach for Structural Data Modeling.” Journal of Computing in Civil Engineering, ASCE, Vol. 6, No. 1, pp. 19-40.

Kolodner, Janet (1993). “Case-Based Reasoning.” Morgan Kaufmann Pub. Co., San Mateo.

Mac Gregor, R. (1991). “The Evolving Technology of Classification-Based Knowledge Representation Systems”, Principles of Semantic Networks, John F. Sowa Editor, Morgan Kaufmann Pub. Inc., San Mateo, pp. 385-400.

Maher, Mary Lou, M. Bala Balachandran, and Dong Mei Zhang (1995). “Case-Based Reasoning in Design.” Lawrence Erlbaum Associates Publishers, Mahwah, NJ.

Rivard, Hugues, Steven J. Fenves, and Nestor Gomez (1995). “An Information Model for Multiple Views of Buildings.” Proceedings of the W78/TG10 Workshop “Modeling of Buildings Through their Life-Cycle, CIB Proceedings, Publication 180, pp. 248-259.

Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen (1991). “Object-Oriented Modeling and Design.” Prentice Hall, Englewood Cliffs, NJ.

Schodek, Daniel L. (1980). “Structures.” Prentice-Hall, Inc., Englewood Cliffs.

Snyder, James, Zeyno Aygen, Ulrich Flemming, and Jonah Tsai (1995). “SPROUT - A Modeling Language for SEED.” Journal of Architectural Engineering, ASCE, Vol.1, No. 4, pp. 195-203.

Wix, J. and D. P. Bloomfield (1995). “Standardisation in the Building Industry: The STEP Building Construction Core Model.” Publication 180, International Council for Building Research Studies and Documentation (CIB), Stanford, California, pp. 184-195.

Zamanian, K. M. (1992). “Modeling and Communicating Spatial and Functional Information about Constructed Facilities.” Phd thesis, Department of Civil Engineering, Carnegie Mellon University, Pittsburgh, PA.