The 5th International Symposium on Robotics in Construction June 6-8, 1988 Tokyo, Japan

Issues in Domain Modeling of Constructed Facilities

Walid T. Keirouz, Daniel R. Rehak, and Irving J. Oppenheim

Department of Civil Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890
USA

Abstract

Robots operating in a constructed facility must possess knowledge about and a description of their environment. This information is organized into a *domain model*. This paper presents current work investigating issues in developing domain models of constructed facilities for robotic applications using an object-oriented approach. Requirements regarding model contents and functionality are identified and the basic architecture of a modeling system is outlined. An object-oriented approach, with *Smalltalk-80* as the implementation language, is used in building the modeling system. Modeling issues under study arise from the descriptions of model objects, relations between objects, and services provided by the modeling system in support of planning activities. Additional modeling issues result from the mapping of the domain into the multi-layered network structure of the model. The paper concentrates on a discussion of these issues and alternative solution approaches to the problems which arise from this object-oriented approach to domain modeling.

1. Introduction

The complexity of an automated construction environment arises from the unstructured nature of the physical environment and from the mobility, versatility, and the ability to alter the environment that characterize construction robots. Tasks in a constructed facility are distributed throughout the environment. Therefore, robots must be mobile and have access to different locations where work is to be performed. Construction activities are diverse, requiring that robots be versatile and capable of performing many different tasks. The domain constantly changes as it is modified by robots and workers, resulting in a highly variable environment. The mobility of construction robots makes it possible for pieces of equipment operating in the same general area to interact and interfere with each other. Finally, the documented description of an environment may differ significantly from the actual facility because unrecorded work may have been performed since the description was last updated.

The use of robots in such a complex domain as construction requires that robots be knowledgeable of their environments. This means that robots must maintain and operate on a computer-based representation of the operating environment. This computer-based representation, denoted a *Domain Model*, reflects the current state of the environment, and is used to generate plans of action, to record modifications made to the environment, to support sensor interpretation, and to retrieve information about the environment that is needed in making intelligent, rational decisions regarding robotic operations. *Domain Modeling* is defined as the development and maintenance of such a domain model.

Studies of several construction and maintenance environments led to the development of a set of requirements that must be satisfied by a domain model in order to support the planning and execution of actions by construction robots. A model architecture was designed to satisfy these modeling requirements, and a prototype domain modeling system was implemented using object-oriented programming techniques. In the process, several modeling issues arose. This paper summarizes the general requirements and architecture of the modal and discusses the modeling issues and details possible solutions to these problems.

2. Domain Modeling Requirements

The complexity of the construction environment makes it imperative to have an explicit domain model. This domain model is defined to be a computer-based representation of the environment. The model exists as a programmed entity, and is maintained separately from the controller of a robot. The controller accesses the model to check the validity of assumptions about the environment and can query the model during planning. The uncoupling of the domain model from the controller of a robot increases the flexibility of the robot and the ease of developing and maintaining programs for complex robotic tasks. The model has a number of uses:

- It can act as a central repository for all information about a facility. A constructed facility such as a power plant contains numerous items and the quantity of information is considerable and wide-ranging in nature. This information varies from system-wide details such as the structural design plans to performance specifications of individual components such as pumps and valves.
- The role of the model as a central repository of data can be extended by having it store a record of changes and modifications in the facility. The model would thus contain an up-to-date version of the known, recorded information about the facility. Planned modifications can therefore be checked against model data to detect actual or potential problems.
- The model can be used to support task planning and execution. A *planner* accesses information in the model to develop plans, evaluate alternatives, and set up contingencies. Physical data collected by the robot can also be matched against expected data extracted from the model to assess the progress of task execution.

- The model can also be used in support of sensor data interpretation. As a robot keeps track of its spatial position, the model can be accessed to yield the set of entities surrounding the location which may match to the sensor data.

The requirements for a domain model arise from its use and can be classified as:

1. what information should the model include so that it can provide an adequate representation of the environment, and
2. what functionality should the model provide in support of planning.

Content Requirements. As a general requirement, a domain model must contain all readily available (i.e., non-computed) information that a planner may need to access. More specifically it must contain:

- *geometric descriptions* of the facility and of entities in it. The geometric description of an entity includes its shape, location, and orientation.
- *nongeometric attributes* of entities in the domain that may be accessed by a planner. Examples of such attributes are color, material properties, weights and names of entities.
- *functionality descriptions* of entities needed to support planning and reasoning. Along with the functional description of an entity, the model must store operative descriptions of basic tasks such as picking up a member or opening a valve.

Functionality Requirements. The set of all possible functionality requirements can be as long as the set of all possible tasks. However, a canonical set of requirements has been generated by considering some characteristics of the constructed facility's domain and a set of critical tasks. Because a domain model must support the planning and execution of robotic tasks, direct support for planning frequently occurring activities and for generating a model of the outcome is beneficial. Specifically, the model must provide:

- *direct support of path planning*. Because robots are mobile, the model should be structured to reduce the cost of path planning.
- *support modification to the environment.* As construction robots modify their environment, the model should be designed to readily permit items and materials to be added to or removed from the description of the environment.
- *inexpensive local updates*. Most modifications in a facility are localized in extent, and thus the model should be "inexpensive" to update locally.

The complete model of a constructed facility will contain a large number of items with considerable detail associated with each item (geometric, non-geometric, and functional) which, must be handled with computational efficiency. Computational efficiency can be achieved by imposing an organization on the model data to reduce the amount of information that must be considered when using the model. Two specific mechanisms are required:

1. *abstract views* that allow a robot to only deal with relevant information. For example, during motion, in traversing a path a robot should only deal with information describing the abstract path instead of descriptions of all entities along the path.
2. *focus-of-attention* that allows a robot to concentrate on its local environment. For example, a robot operating on a valve should only be concerned with the area around the valve and not with details at another location.

3. Domain Model Architecture

The requirements presented above stipulate that the domain model must contain comprehensive information describing the entities in the facility, and that it must organize the data as to provide abstract views of the model in support of its various uses. Based on these requirements, a domain model can be viewed as representing a constructed facility at two levels:

1. At the *macroscopic* level, the model imposes an organization on the information describing the environment. This organization provides the various levels of abstraction of the domain that support the functional requirements imposed on the model.
2. At the *microscopic* level, the model represents individual entities. It includes all the details, descriptive and functional, about all entities in the domain.

These views translate into an overall structure of the model and its components.

Model Structure. At the macroscopic level, the architecture of the domain model has a hierarchical structure with a network-based representation. At the top level, a facility is represented as a single model object. This model object provides a general description of the facility and holds properties (such as geographic location and overall dimensions) that apply to the entire facility. The model object representing the entire facility is decomposed, along spatial boundaries, into subobjects corresponding to parts of the facility. A more detailed representation of the facility is obtained by considering lower levels in the hierarchy, where subobjects hold properties that are specific to the parts of the facility they represent. The recursive decomposition of objects into subobjects can be carried out until objects standing for non-decomposable physical entities are reached.

Subobjects resulting from the decomposition of an object into its components are organized into a network. Subobjects in the network are represented by the nodes in the network while the branches model spatial and functional relations between subobjects.

A bottom-up traversal of the hierarchical structure of the model reveals the mapping from physical entities onto model objects. At the bottom of the hierarchy, non-decomposable physical entities are mapped onto objects without subobjects. At higher levels,

complex physical entities have corresponding model objects whose subobjects represent the components of the entity. Relations between physical entities are represented by relations between the model objects corresponding to these entities.

The mapping from entities onto objects also occurs at the level of functional systems. At the physical level, entities can be organized into the functional systems (e.g., electrical, piping, and structural) to which they belong. Models of functional systems can be assembled from objects representing entities with the appropriate functional affiliations, and thus will correspond to their counterparts in the facility.

An object representing a complex physical entity can also organize the empty space enclosed within its boundaries into a network of *virtual spaces*. The organization of the space unoccupied by the components of a physical entity can be used to specify frequently used paths through the space.

Model Objects. Individual model objects are defined at the microscopic level. The model architecture is characterized by the use of an *object-oriented* framework for defining individual objects. Each model object is an independent and self-contained unit in the model, and objects which represent a physical entity contain all the descriptive and functional information describing that entity. For example, the object modeling an electric pump contains: a geometric description of the pump (e.g., its shape, location, orientation); a description of the non-geometric properties of the pump (e.g., its maximum flow rate, head increase); and set of operators for manipulating the pump (e.g., turning it on or off).

A complete domain model is built from four basic types of objects:

1. *Primitive objects* occupy terminal or leaf nodes in the object hierarchy. They have no subdivision and represent non-decomposable physical entities, such as pipes, pipe hangers, and structural members.
2. *Domain objects* are the non-terminal or intermediate nodes in the object hierarchy. They are recursively built from other objects and have a network decomposition. Domain objects represent complex physical entities made of several components such as pumps, or entities that define space enclosing other physical entities such as rooms.
3. *Connection objects* are the branches in the network decomposition of domain objects. They represent spatial and functional relations and connectivities between objects.
4. *Virtual space objects* represent unoccupied space enclosed within the boundaries of a domain object. The virtual space objects are organized into a network and can be used for path planning.

Prototype Modeling System. A prototype domain modeling system based on an object-oriented programming methodology has been implemented in *Smalltalk-80*. The object-oriented model of a facility uses the object-oriented programming methodology as a paradigm for representing and organizing the components of the model. All components of the domain model are objects in the object-oriented programming sense: each is a self-contained, independent computational entity with a private memory for storing its attributes and code (e.g., *methods*) that define operators for manipulating its attributes.

The model integrates descriptive information (quantifiable characteristics which are stored in the attributes of an object) with functional information (embedded in the operators of the objects). The object-oriented programming approach provides a flexible and extensible model development environment, and uses inheritance mechanisms to build new object and special capabilities on top of those provided by the basic modeling system.

Note that the modeling system does not contain any higher level functional capabilities. Rather, each user of the model (e.g., a planner) uses the basic functional capabilities provided to implement its own functionality.

4. Modeling Issues

While the basic model architecture can be justified through its object-oriented approach, and through fufillment of all the imposed requirements, several major modeling issues have been encountered in developing the prototype. These issues arise from the descriptions of model objects, relations between objects, and services provided by the modeling system in support of planning activities. Additional modeling issues result from the decomposition of the domain into the multi-layered, spatially oriented, network structure of the model. These issues are discussed below.

4.1. Geometry Services

Other than the capability to represent geometric attributes of primitive objects, the geometry of a facility is not explicitly represented in the domain model. Rather, geometric properties and relations must be derived, on demand, from this primitive data which represents the overall structure of the facility through the hierarchical decomposition of spaces.

Any higher level problem-solving capabilities supported by the model which require geometric reasoning (i.e., task planning) must be based on lower level *geometry service* primitives. If these primitives do not exist in the basic model architecture, they must be provided by each user of the domain model, but this results in redundant development efforts. As a solution to this issue, a set of basic geometry services is included in the model to support planning activities.

,prehensive treatment of support for geometric reasoning is beyond the scope of this work, and the field of geometric ,ning is still an active research area. Rather than attempt a comprehensive solution to the problem, the model architecture has ,n extended to provide basic geometry services that can be used as building blocks for developing more complex geometric ,asoning. This extension illustrates that the model can be successfully applied in tasks where geometric reasoning is required.

Reflecting the object-oriented paradigm underlying the model, the geometry services are provided by individual model objects that represent physical entities. These objects answer geometry queries issued by a planner. The geometry services are provided at two levels: primitive objects respond to queries about their own geometric description; domain objects service queries about the geometry of their subobjects and about the space a domain object encloses.

Primitive Objects. A planner needs to have access to the geometric description of individual objects, i.e., its location, orientation, (defined with respect to a specific anchor point), and shape (e.g., a cylinder, cube, or sphere). In addition, a mechanism must exist to respond to the following questions about the geometry of a primitive object at an abstract level:

- What is the object's outer shell (smallest enclosing envelope)?
- What is its smallest bounding box with a given orientation?
- What is the object's proximity to or contact with another object?
- Can the object be enclosed within another object,

Facilities for all of these are provided as basic components of the model.

Domain Objects. Primitive objects have a local self-centered perspective on geometry, but the planner may need to deal with the geometry of several objects at the same time. Such a perspective is provided by the domain objects via their responses to the queries.

In support of path planning, domain objects service two categories of motion queries: simple motion queries which ask a domain object if one of its subobjects can be moved using single step motions (i.e., translation or rotation), and complex motion queries built on the simple queries. Complex queries include asking a domain object whether one of its subobjects can follow a given path, whether a path exists for moving a subobject from one location to another within the domain object (and if so, what the path is). To aid in computational efficiency, the path that is computed in order to respond to a query is retained so the path does not need to be recomputed if requested again.

A planner also needs to access information about the space and objects within a domain object, such as the movability of a subobject, the existence and identity of an object occupying a certain location, the existence and identities of objects filling some given space, the distance from a subobject to a location, the identity of the object nearest to a location, and the set of objects within a specified distance from a location.

Support for additional queries which involve multiple domain objects are also required. Examples of such queries are determining the set of objects globally within a distance from a point and the object globally nearest another object. In order for the responses to these queries to be meaningful, the answer should be the objects having the same level of abstraction (e.g., the object globally nearest a room should be another room, not some object within that other room).

The set of geometry services outlined above is included in the domain modeling system and is based on a review of the needs of a planner, but is not unique or complete. However, it provides a range of services that can be used as basic blocks for answering more complex queries.

4.2. Inheritance

This section discusses three cases where an inheritance-like relationship must be considered or formalized:

1. *class inheritance.* An object is defined by its class and all of its superclasses. But from this hierarchy, it is not obvious what the proper defining type of an object is. For example, "*pump_#17*" is defined by several classes which include "*Pump*" and "*Electrical_Device*". Is "*pump_#17*" a "*Pump*", an "*Electrical_Device*", a "*Primitive Object*", or is it best defined as some other type?
2. *inheritance from attributes.* An object's properties depend on the types of it attributes, and these attributes influence the object's behavior. For example, pumps may have different shapes (e.g., cylindrical and spherical). Pump objects will respond to different queries depending on the *shape* attribute. Given this situation, how does an object inherit behavioral characteristics from its attributes?
3. *inheritance from functional systems.* An object which is a part of a functional system has its behavior modified as a result of its functional system affiliations (e.g., an electric pump will operate only when a part of both an electrical and a piping system). What mechanism should be used for an object to inherit behavioral characteristics from its affiliated functional systems?

Class Inheritance. The issue of class inheritance arises when trying to determine the type of an object in order to reason about it. The inheritance structure organizes classes into a hierarchy if a class can only have one superclass, or into a graph when multiple

superclasses are allowed. Through inheritance, an object acquires its type and properties from its own class and from the inheritance hierarchy above its class. Thus, the type of an object can have multiple values. While an object is an instance of its class only, it may be appropriate to consider it as an occurrence of one of its superclasses.

One mechanism used to determine the type of an object is to set up a dialogue with the object. When the type is requested, the object first answers with its most specific type, namely its class. Through an iterative process, the inheritance graph is traversed, and more general types are returned on each subsequent query.

However, not all classes in the inheritance graph can be considered as responses. Because there are classes in the inheritance hierarchy which do not define the behavior of an object, these classes must be bypassed when traversing the hierarchy.

- An *instantiated* class (i.e., "*Electric_Pump*") defines the actual behavior of its instances by specializing other classes and is therefore an acceptable object type response.
- An *abstract* class (e.g., "*Pump*") defines the general properties of families of similar objects but does not have instances itself. An abstract class is specialized into an instantiated subclass and thus is an acceptable type response for reasoning about objects.
- An *internal* class is used to implement an aspect of a data structure or a behavioral trait that is inherited by classes beneath it. It does not correspond to an entity in the real word, and thus is not an acceptable object type response.

Because object-oriented programming systems do not distinguish between class categories, the domain modeling system must be extended to provide the necessary functionality to determine an object's type. This is straight-forward; a class instance variable is used to record the class category, and because classes are objects themselves, methods are defined for the class objects to properly respond to queries about object type.

Besides the issue of the appropriateness of various classes as responses, the order used to traverse the inheritance graph is problematic. If only single inheritance is allowed, the inheritance graph is a tree and can only be traversed from the object to the root in a unique order. However, if multiple inheritance exists, the inheritance graph can be traversed *depth-first*, *breadth-first*, or in a combination of depth-first and breadth-first order. An explicit traversal scheme is usually built into an object-oriented programming system (e.g., *Smalltalk-80* provides depth-first, but includes the capabilities to implement other traversal schemes).

A traversal scheme such as depth-first is inadequate as it leads to the loss of specificity when trying to determine the type of an object. Consider an "*Electric_Motor*" which is defined as a subclass of both "*Electrical_Device*" and "*Mechanical_Device*", both of which are of type "*Domain_Object*". In depth-first traversal, *Electric_Motor* is first considered an *Electrical_Device*, and subsequently considered a *Domain_Object* before it is considered a *Mechanical_Device*. However, it is more meaningful to consider it as a *Mechanical_Device* before considering it as a *Domain_Object*. Therefore, breadth-first is viewed as the more appropriate traversal scheme, and a modified graph traversal algorithm which handles multiple inheritance and skips internal class nodes has been implemented in the modeling system.

Inheritance from Attributes. Consider the case of an object that has attributes which are modeled by objects (e.g., the shape of a pump is an attribute modeled as a shape object). Inheritance from attributes is a mechanism that allows an object (e.g., pump) to automatically delegate the responsibility of answering certain queries to its attributes (e.g., the shape object responds to shape queries). In the object-oriented framework, this type of capability does not exist. Because inheritance from attributes is useful in modeling, failure to provide this capability can be detrimental as illustrated below.

When attempting to find out about details of a pump's geometry, the planner would recognize that the pump has a shape attribute. Based on the type of shape (e.g., cylinder, sphere), the planner could then directly query the appropriate shape object to determine geometric information (e.g., radius and length versus diameter). To do this, the planner would need to know about all of the possible pump shapes, and the proper mechanisms for retrieving details of each. The planner would be dealing with the attributes at a detailed level, rather than at an abstract level. Thus, ignoring attribute inheritance is unacceptable because it delegates too much responsibility to the user of the model.

A simple alternative is to specialize the inheriting object with a set of subclasses to deal with each possible class of an attribute. Thus, in the example of cylindrical and spherical pumps, the classes "*Cylindrical_Pump*" and "*Spherical_Pump*" are defined as subclasses of "*Pump*" and they can respond to queries about their specific types. This approach requires that appropriate subclasses be defined in all situations on a case by case basis, and is feasible only when the number of attribute types is small. A general solution to the problem would be preferable.

General methods for solving the problem are based on automatically modifying the classes and objects in the system to handle the inheritance. One alternative is similar to passing the responsible of dealing with inheritance to the user. When the value of an attribute is set to be an object, the modeling system automatically creates a mechanism to forward messages from the object itself to the appropriate attribute object. In *Smalltalk-80*, this is implemented by specializing the object's "*doesNotUnderstand:*" method to include a table of all of the methods to be inherited from all of the object's attributes. Because the object itself does not know what requests the attributes can handle, when the object receives a request which should be handled by one of the object's attributes, the object will not recognize it. The "*doesNotUnderstand:*" method is invoked, searches the table of all methods it understands, and forwards the request to the appropriate attribute object. This approach, while conceptually simple, is computationally inefficient in that the forwarding mechanism is established for each object, rather than for classes of objects, and the search and forward mechanism has significant overhead.

ely, a mechanism can be developed to automatically specialize the inheriting object with a set of subclasses. This concept cal to the first alternative illustrated above, but instead of the user defining all of the appropriate subclasses, the modeling n can be extended to automatically build new specialized subclasses on demand. The specialized subclasses define methods correspond to the methods to be inherited from the attribute and automatically forward all message to the attribute if appropriate. For example, when a pump object is defined to have a cylindrical geometry, a new "*Cylindrical_Pump*" class is automatically created (if it does not already exist) which contains the attributes of both the pump and cylinder objects, and the pump object is automatically redefined as a cylindrical pump object. While this approach is more efficient than the previous one, defining methods that can automatically specialize a class and change the instance of an object from one class to another is very complex.

Rather than implement this last mechanism, the scheme based on forwarding messages has been incorporated into the modeling system. This mechanism appears to meet all of the requirements, is easy to implement, and the inefficiencies are not of concern in building the prototype system. The implementation could be replaced by the last mechanism described to gain efficiency without a loss in functionality.

Inheritance from Functional Systems. Objects can be components in functional systems (the issue of representing functional systems is discussed below), and the behavior of an object is affected by being a component in a functional system (e.g., pumps can operate only if connected). Inheritance from functional systems is the mechanism through which an object's behavior is modified on the basis of its functional affiliations. Two mechanisms exist for an object to inherit functional behavior from its associated functional systems.

The first mechanism provides for an object's behavior to change when it becomes a component of a functional system and inherits the characteristics of the system (i.e., certain behaviors are undefined if the object is not connected to a functional system). For example, a pump does not know it can impart momentum to a fluid until it is connected to a piping system. This mechanism does not allow entities affiliated with more than one functional system to be used to couple the functional systems, because the mechanics of the coupling depend on the type of the coupling object, but these behavioral characteristics must be inherited from functional systems.

Alternatively, the behavior can be associated with the object and the behavior is activated only when the object has a functional affiliation. For example, a pump would know it can impart momentum, but knows it can only do so when connected to the piping system. The object would also know about all its potential functional affiliations. If a class which describes common properties and behaviors of all components of a functional system is defined, objects that are potential components of the functional system can be a member of this class and thus directly inherit the behavior of the functional system. When interacting with a functional system, the object need only determine if it is connected to the functional system. Completing the connection activates the behavior.

At this time all of the issues associated with inheritance from functional systems have not been worked out. Thus, beyond this general discussion of the problem, additional issues, recommendations, and details of a solution strategy can not be presented.

4.3. Class Organization for Families of Entities

Class organization becomes as issue when modeling physical entities which are members of the same family of objects. Such entities share basic properties, but can be of different types and have different behaviors. For example, pumps serve the same functional purpose and share the same properties (inlets, outlets, imparting momentum to fluids), but pumps come in a variety of models and types. The classes of these model objects must be defined to take advantage of the similarities among the objects they describe while accommodating their various particular behaviors. The classes must also be organized to maximize both the efficiency of data access and the ease and flexibility with which new types of entities can be defined within the same family.

Two basic schemes can be used to define classes of families of objects: shallow and deep hierarchies. In a deep hierarchy, numerous levels of abstract and non-instantiable classes define the properties common to all members of a family. A class specializing the abstract family class is defined for each subfamily of entities, and class specialization is repeated until a class is defined for each type (model) of entity. Common properties are placed at the highest possible level in the hierarchy. Only the lowest-level type-specific classes are instantiable, the abstract family and subfamily classes are not. For example, the properties common to all pumps are defined by the class "*Pump*" which is not instantiable, but which is recursively specialized until a class is defined for each pump model ("*Electric_Pump*", "*Continuous_Action_Electric_Pump*", "*Centrifugal_Electric_Pump*", "*Centrifugal_Pump_Model_XYZ*"). Of the five classes, only the last class ("*Centrifugal_Pump_Model_XYZ*") is instantiable and its instances stand for pumps of that model. Organizing classes in a deep hierarchy may make it difficult to determine where to add new types of entities into the hierarchy, as the decomposition and classification of objects into the levels may be somewhat arbitrary, and adding a new class may require modifications and additions in several levels of the hierarchy.

The alternative scheme organizes the classes of a family of entities into a shallow hierarchy. As in the first scheme, an abstract, non-instantiable class defines the properties common to the family of entities being modeled. This family class is directly specialized into an instantiable class for each subfamily of entities. The differences between entity types are handled by storing type-specific methods in the instances to tailor the generalized methods defined in the subfamily classes so that they represent the individual entities.

For example, only two classes, "*Pump*" (family class) and "*Electric_Pump*" (subfamily class), are needed to model a specific centrifugal pump, "*Centrifugal_Pump_Model_XYZ*". This pump is as an instance of the subfamily "*Electric_Pump*". This

instance contains all the information that identifies the behavior and components of the specific pump being modeled, and distinguishes that pump from a generic electric pump. It may be difficult to model all of the differences between an individual instance and the generalized class by storing appropriate parameters and methods in the instance as the parameters may not be easily identifiable and the generalized methods difficult to write. This organizational scheme also exhibits redundancy as information stored in instances is duplicated when pumps of the same type are modeled. One potential solution to the redundancy problem uses prototype objects which store only the differences between classes. However, the use of prototypes further complicates the distinction between classes and instances.

A hybrid of the two organizational schemes described above can be developed that combines a deep hierarchy with type-specific information stored in the individual entities. However, additional research is needed to identify the most flexible scheme.

It should be noted that the issue of class organization is not of import when developing prototype models of simple domains. Such models deal with a limited number of physical entities and with an even smaller number of entities from the same family. Thus a solution whereby each entity type has a corresponding class is acceptable. Due to the scope of the prototype modeling system, the first alternative (a deep hierarchy) has been implemented. However, the class organization issue is important from a conceptual point of view as it affects the modeling of individual entities and the type of information stored in each object. The other alternatives outlined would be more appropriate in more complex domains, but would still provide the same features as the solution which has been implemented

4.4. Representing Assemblies

The issues of modeling assemblies and accessing assembly information arise when complex objects such as pumps must be repaired. A pump typically is modeled as a single physical unit. However, it is composed of a collection (an *assembly*) of other entities: impeller, casing pieces, motor, shaft, bearings, nuts and bolts, etc. A robot repairing such a pump must know how to take the pump apart and reassemble it, and someone must keep track of the various components of the pump while it is disassembled.

An assembly modeled as a single domain object after being disassembled becomes a collection of more basic entities represented as primitive and domain objects in the model. In modeling an assembly, several issues must be considered: assembly and disassembly information must be accessible independent of the status of the assembly; a mechanism for identifying and keeping track of the components of an assembly entity is needed; the modeling system must be able to realize when an assembly has been reassembled; and the replacement of a component of an assembly by a new part must be a feasible operation.

There are two basic alternatives for modeling an assembly. In the first alternative, all relevant information is stored in a domain object which represents the assembly and contains all of the information about assembly and disassembly operations as well as information about parts of the assembly. Because the assembly object exists independent of its physical realization, it can be used to model the transitional state when it does not represent a physical entity, but rather an entity being assembled or taken apart. This solution handles all the relevant issues identified above. However, it can suffer from redundancy of information if information common to multiple objects of the same type must be stored in each model object (e.g., instances of the same model pump all having duplicate assembly information).

In the second alternative, all of the relevant information common to all instances is stored in the classes of domain objects representing the assemblies (instead of in the objects themselves). The class of an object representing an assembly contain information about assembly and disassembly of the corresponding physical entities. When a physical object exists, an instance of the class exists which contains a list of the assembly's subobjects. This alternative suffers several deficiencies. Because the class does not differentiate between its instances, assembly information is given in a parametrized form by specifying the classes of the parts instead of their identities. Furthermore, since the model does not contain a mechanism for keeping track of the parts of a disassembled assembly, this role must be assumed by the user of the model. In addition, an assembly which is disassembled and then reassembled will be represented as different objects. These problems can only be remedied by developing a complex mechanism for maintaining the parts and maintenance history of the assembly.

A third alternative is based on a combination of the two cases outlined above, combining the benefits of the two alternatives while correcting their deficiencies. The assembly and disassembly information is stored in both model objects (for specific information relevant to only a single object) and their classes (for information common to all instances). A class contains the plans for assembly and disassembly, as in the second case. The assembly object exists independent of the object it represents, so that it can model transitional states, as in the first case. The assembly object also keeps track of the identity of its components by maintaining the list of its subobjects.

This third alternative is used in the prototype modeling system. Assembly information is added to the definitions of models and classes as they are implemented. This current implementation is adequate for the prototype system. However, it would need to be refined for a system using a shallow class organization of families (described above), as the distinction between classes and instances partially disappears when a shallow class organization is used.

4.5. Connection Objects and Inter-Object Relationships

A characteristic of the modeling scheme is that each physical entity in the domain is represented by an object in the model. Inter-object relationships in the domain are represented by connection objects which model:

1. *geometric relations* such as proximity and contact, and
2. *functional relations* whereby entities in the domain form combine to form part of a functional system (e.g., connected pipes in a piping system and interlocking parts in a mechanical system).

Modeling of connectivity, however, is complex and requires that the domain modeling system incorporate several capabilities so that it can fully represent connectivity.

A connection object is limited to representing the relations between only two objects, a restriction which results in a canonical representation without limiting the modeling capabilities of the proposed scheme. A connection object has features to model both the geometric relations and the various functional system affiliations that the two objects share.

It should be noted that a coupling exists between the functional and geometric facets of connections. At a primitive level, a functional link cannot exist between two objects representing physical entities without the entities being physically connected. The connection between the objects implies that a geometric link also exists between the two objects.

The modeling system must address the following issues:

1. Because functionally related entities in a domain belong to functional systems (e.g., electrical, mechanical, or piping system), capabilities for modeling functional systems and establishing functional links between objects must be present in the model.
2. Because connections between objects exist not only as abstract relationships, but also as physical links, and since a physical object can act as a connection, it may be desirable to explicitly allow connections to possess both a physical reality and a functional reality and to represent these physical domain entities as connection objects.

Functional Systems and Functional Affiliations. Using the geometric decomposition of the domain to organized the domain is adequate for geometric information, but it does not provide a natural mechanism for organizing information about functional system affiliation. Modeling of functional system affiliations is further complicated because a single entity can have multiple affiliations. For example, a piping system consists of pumps, pipes, and valves, linked by piping component connections (which are both abstract connections and real objects), but an electric pump in the piping system is also a part of the electrical system. Two organizational schemes that deal with modeling this functional affiliation information can be described as *implicit* and *explicit* schemes.

In the implicit scheme, the existence and constituents of a functional system are inferred from inter-object relationships. Modeling of the behavior of a functional system is handled by having objects transmit information along the functional links of the system (i.e., the behavior is modeled by having the components of the system simulate their own behavior and communicate the results to their neighbors). For example, components of a piping system are informed of the opening of a valve when the valve communicates the change in pressure to the downstream pipe via a connection object. The pipe then propagates the information downstream to the next pipe, and the process is repeated throughout the system until the pressure in the system reaches equilibrium.

The implicit scheme is consistent with and readily implementable in the object-oriented paradigm, wherein communication between objects is performed by sending messages. But because functional systems are not explicitly represented, information about such systems must be generated when needed by traversing all of the links that represent the functional system. Furthermore, this scheme restricts processing to the use of relaxation or network propagation techniques. Use of a direct solution technique would require the generation of an explicit representation of the functional system. Thus modeling of functional system behavior may require complex and expensive operations.

In the explicit scheme, a functional system is represented as an object in the model, and the components of the system are linked to this object. Processing is not limited to the use of relaxation techniques. Using direct solvers, an object first informs the functional system with which it is affiliated of a change. Using the behavior model associated with the functional system, the state of all components in the system is determined, and the necessary information is communicated to all of the components which comprise the functional system. For example, when the valve is opened, it informs the functional system object representing the piping system which directly computes the pressure and flow rate in the entire piping system, and then informs each component of its new pressure and flow rate.

In addition to the problem of how to represent functional system affiliations, there are the issues of how and when to establish a functional link between objects. These issues arise when components must be assembled or disassembled. For example, when a pipe is connected to another pipe by a leak-proof connection, the pipe becomes a part of a piping system. One way to establish the functional affiliations is to require the user of the modeling system to maintain these connections in parallel with the modifications performed on components of the system (e.g., the assembly). However, it is preferable for the modeling system to automatically create functional links as required. This process may be organized in a *centralized* or a *distributed* manner.

In the centralized scheme, the criteria used to determine when a functional system affiliation exists are stored with the specific functional system, via functional system affiliation objects. Each model object is declared to be a part of one or more types of functional systems. Whenever a physical connection is established between two objects in the model, the functional system affiliation objects associated with all potential functional system affiliations common to both objects determine and set up connec-

tion objects if the criteria for establishing functional links are satisfied. This scheme is complex because the functional system affiliation objects must be able to handle all of the different cases that lead to the establishment of a functional link (e.g., all the ways to connect two pipes in each type of piping system).

Alternatively, the information needed to establish functional links may be stored in the primitive and domain objects that represent physical entities. Individual objects know what criteria are necessary to establish a functional connection, and create a functional link whenever these criteria are satisfied. This distributed scheme is consistent with the object-oriented paradigm. However, there is a consistency problem as all objects which can be connected must consistently represent the equivalent sets of requirements for creating a functional link.

Functional systems are explicitly represented in the prototype modeling system, and the distributed scheme is used to organize the functional affiliation criteria. The consistency issue is unresolved at this time, and the model developer is responsible for maintaining consistency.

Physical Reality of Connection Objects. A characteristic of the model architecture is that connection objects in the model stand for relations between domain entities. However, some entities, such as pipe connectors, might be modeled as physical objects but also function as physical connections. There are two alternatives for representing such entities. They can be modeled using the existing constructs of primitive, domain, and connection objects, or the definition of connection objects can be extended to allow physical entities to act as connections.

Following a strict interpretation of the model architecture, a connection object would be a purely relational entity without a physical reality of its own. The only physical information represented in a connection object is the point of connectivity. A physical entity functioning as a connection is represented as a primitive or domain object in the model. The model object together with its connections represent the functionality of the physical connection entity. Characteristics of this modeling scheme include simplicity and the existence of a one-to-one mapping from entities in the domain to objects in the model. However, this simplicity makes it impossible to explicitly represent the functionality of physical connections. Hence this functionality must be inferred from the model. This is not necessarily a trivial task. Consider the bolted connection between two pipes which require a gasket to render it leak-proof. The two pipes are in contact with each other and with the gasket. Modeling the physical connection requires one connection between the pipe and gasket and another between the gasket and the second pipe. No model object explicitly represents the piping system link between the two pipes. Rather, the existence of a piping system connection must be synthesized from the behavior of the connection objects.

The second modeling alternative removes the restriction that connection objects be purely relational entities. As such, connection objects can have a physical reality, represented by an associated primitive or domain object. Thus, a physical entity acting as a connection in the domain is directly mapped onto a connection object with a physical reality reflecting that of the entity. For instance, the connection object linking two pipes can have a gasket as its physical reality and explicitly represents the piping system connection between the two pipes. However, this modeling alternative does not maintain a canonical decomposition of a domain.

In the current implementation, connection objects follow the strict interpretation of the model architecture and are only relational entities. A variant of the second alternative is being developed. This version restricts the types of physical entities than can act as connections, and attempts to maintain the canonical decomposition of the environment whenever possible.

4.6. Entities at Domain Object Boundaries

As noted, the domain modeling system decomposes the environment into a hierarchy of entities based on spatial partitioning. Entities at domain object boundaries cannot be easily placed in the hierarchy of model objects because they do not fit well in this canonical structure. This problem arises in several situations:

- An entity can exist at a location corresponding to the boundary between two domain objects, and can occupy space represented by the two model objects without being completely enclosed by either of them. For example, a double swing door exists at the boundary between two rooms represented as domain objects, but the door can intrude on the space of either two rooms without being part of either.
- A single entity can traverse the space represented by several domain objects, but the object representing the entity cannot be modeled as a subobject of any of the domain objects it traverses. For example, a pipe is a single entity (which is not logically decomposable) which can traverse several rooms, each modeled as independent domain objects.
- When an entity is being moved, as it crosses from the space of one domain object to another, the entity occupies space in both domain objects.

An entity spanning the boundary between two domain objects must appear to be part of both domain objects. The simplest solution consists of allowing an object to be the subobject of more than one domain object, but this violates the principle that a subobject is geometrically enclosed within its parent object. Thus a special mechanism must be developed for representing an entity that crosses the boundaries of domain objects.

The basic solution mechanism is to represent the entity as a subobject in each domain object and to constrain these two subobjects to have equivalent properties to ensure that the two model subobjects refer to the same physical entity. This equivalence can be maintained by the use of constraints and constraint propagation techniques; when an attribute of one of the subobjects is changed,

constraint propagation is used to inform the other subobject. However, simple constraint propagation of value equality is insufficient in certain cases. For example, if one end of the subobject is moved and translated in one domain, the part of the subobject in the other domain moves by a different amount.

Alternatively, entities which cross domain boundaries can have associated spatial mappings. These mappings are used to represent the spatial characteristics of the physical entity in each domain object without mapping other properties of the entity into the domain. Thus the domain objects include subobjects which only represent the physical presence of the entity, and the entity itself is a subobject of some higher level domain object which completely encompasses it. This solution includes a mechanism so that messages sent to the spatial subobjects are forwarded to the corresponding physical entity. This alternative appears to be adequate and is readily implemented in *Smalltalk-80*, and thus is used in the prototype.

5. Closure

While a number of significant modeling issues have been raised, none of these appear to seriously challenge the overall model architecture. Rather it appears that adequate solutions exist for all of the problems described. The resulting prototype modeling system demonstrates these solutions and the validity of the domain model architecture.

It must be noted that the set of issues presented is not necessarily exhaustive, but rather is based on experiences in investigating several domains and in building the prototype modeling system. Thus additional problems and issues may arise when trying to extend the use of the model, when applying it in other domains, or when using the prototype's architecture as the basis for the design of a production system. In addition, issues such as direct model support for sensor interpretation, support of path finding, and support for task specification, problem solving and planning are still under investigation.

The conceptual solutions to the issues discussed above are independent of the programming environment used to implement the modeling system. While *Smalltalk-80* has been an adequate implementation environment, other object-oriented programming systems provide different capabilities which will influence the details of the modeling system. Some object-oriented programming systems may offer built-in features which would simplify development, but such differences impact only details of the implementation, and have no impact on the model architecture which has been developed.

The general object-oriented approach has had a significant impact on both the conceptual evolution of the domain model and the detailed implementation of the modeling system. Overall, the object-oriented approach to developing the domain model has been intuitive and natural. In many cases, the object-oriented nature of the model and the parallels between physical entities and their models as objects directly led to a solution to an issue. The object-oriented programming implementation made it easy to change portions of the model and the prototype without impacting other portions of the system. Thus the object-oriented programming language approach provided the encapsulation and abstraction necessary in implementing a large software system. While the object-oriented programming approach is not without problems (implementing some of the necessary structures and facilities in *Smalltalk-80* has been complex), overall it appears that such an approach is a superior mechanism for software development. In addition, the philosophy and conceptual structure of an object-oriented system simplified the overall development of the domain modeling system.

Acknowledgments: This work was supported in part by the Electric Power Research Institute, Project 2515-1; by a U.S. National Science Foundation Presidential Young Investigator Award, Grant number ENG-8451533; and by Cray Research.

References

[1] Keirouz, W.T., Rehak, D.R., and Oppenheim, I.J., "Object-Oriented Domain Modeling of Constructed Facilities for Robotic Operations," *Proceedings, First International Conference on the Application of Artificial Intelligence in Engineering Problems*, Vol. 1, D. Sriram and R. Adey, Eds., Southampton, England, Springer-Verlag, Berlin, pp. 141-150, April, 1986.

[2] Keirouz, W.T., Rehak, D.R., and Oppenheim, I.J., *An Object-Oriented Programming Approach to the Development of Computer-Aided Engineering Systems*, Technical Report EDRC-12-09-87, Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[3] Keirouz, W.T., Rehak, D.R., and Oppenheim, I.J., "Development of an Object-Oriented Domain Model for Constructed Facilities," *Artifical Intelligence in Engineering: Tools and Techniques*, D. Sriram and R.A. Adey, Eds., Second International Conference on the Applications of Artificial Intelligence in Engineering, Boston, MA, Computational Mechanics Publications, pp. 259-271, August, 1987.

[4] Keirouz, W.T., *Domain Modeling of Constructed Facilities for Robotic Applications*, unpublished Ph.D. Dissertation, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[5] Oppenheim, I.J., Rehak, D.R., Keirouz, W.T., and Woodbury, R.T., *Robotic Task Planning: Domain Modeling and Geometric Reasoning*, Technical Report NP-5525, Electric Power Research Institute, Palo Alto, CA, December 1987.

[6] Woodbury, R.F., Keirouz, W.T., Oppenheim, I.J., and Rehak, D.R., "Geometric and Domain Modeling for Construction Robots," *Proceedings, International Joint Conference on CAD and Robotics in Architecture and Construction*, Marseilles, France, June, 1986.