ThingLab — An Object-Oriented System for Building Simulations Using Constraints

Alan Borning
Learning Research Group
Xerox Palo Alto Research Center
Palo Alto, California 94304

Introduction

ThingLab is a system that provides an environment for building simulations. Within this environment, a simulation is organized as a collection of *objects* that interact with one another by sending and receiving *messages.* Each object has a table of named *properties,* which describe both the internal state of the object and its protocol for sending and receiving messages. An object may have one or more *parents,* thus providing for inheritance of properties. The interactions of the objects in a simulation are specified in terms of *constraints.* The operation of *merging* is used to specify connectivity and to combine properties inherited from several parents.

The domain of the system is the micro-world of simple physics experiments. A user can construct simulations of parts that interact in mathematically well-defined ways; examples of things that can be simulated are electrical circuits, bridges and truss networks, digital logic circuits, and mechanical linkages. The system has no built-in knowledge of electricity or bridges, but rather provides tools that make it easy for a user to construct objects that contain such knowledge. We plan to explore ThingLab as a vehicle for teaching some of the concepts of physics.

Previous Work

One of the principal influences on the design of ThingLab has been Sketchpad [Sutherland 1963]. Many of Sketchpad's interactive graphics techniques have been adopted, but its domain of constrained geometric objects has been extended to include more complex simulations such as electrical circuits. The Sketchpad notions of constraints and of recursive merging have been used extensively.

The other main influence on the design has been Smalltalk [Learning Research Group 1976. Goldberg A Kay 197(>]. Smalltalk is a simulation language in which knowledge is organized around the conceptual objects involved in the program, rather than around a set of procedures that act on data. Objects communicate with each other by sending and receiving messages. *Classes* describe the common properties of sets of objects, while *instances* represent individual objects. Ideas from other object-oriented languages, in particular Simula [Dahl. Myhrhaug. & Nygaard 1970], KRL [Bobrow & Winograd 1977], and the various Actor formalisms [Hewui 1976], have also been very useful.

The Smalltalk notion of an object has been modified and extended in a number of ways. In ThingLab, objects are constructed interactively by editing and making descendants of prototypes. An abstraction hierarchy is used that allows arbitrarily many levels of both parents and descendants, with the problem of objects with several parents being handled by the technique of merging. There is no distinction between classes and instances. In Smalltalk, the description of the way an object receives a message is stored only in procedural form; in other words, it is the list of statements that are evaluated to receive the message and generate a reply. In ThingLab. however, an action such as *show* or *print* is implemented as an object in its own right. This allows the way in which an object shows or prints itself to be described in a declarative manner, and also allows an object with several parents to construct an appropriate action by merging the corresponding actions from its parents.

ThingLab Objects

Each object has a *table of properties* indexed by *names.* Each property is in turn another object. The properties of an object describe both its internal state and its protocol for sending and receiving messages. An object may have one or more *parents,* each of which is also another object

An object receives a message in the following way. The object fetches the first token, which must be a property name, and looks it up in its table of properties. If the name is not found, the object asks its parents to look up the name. The parents may in turn ask <u>their</u> parents to look up the name, and so on. If the object has several parents, and more than one parent returns a property, a new property is constructed by merging the inherited properties (see below). After the property corresponding to the name has been found or constructed, the property is evaluated in the context of the object which received the original message, and the result of the evaluation is applied to the remainder of the message.

This scheme requires that property names be used in a consistent way. For example, each property named *show* should be a descendant of *Picture,* and each descendant of *Picture* should know how to merge itself with another such descendant If the same property name is used coincidentally by two parents for quite different things, the two properties can't merge, and the user will be asked what to do.

To allow one object to serve as a property of another, each object has associated with it a piece of code which is evaluated to receive a message for another object. For most objects, such as real numbers or points, this code just returns the object itself. For example, the *resistance* property of a resistor should be a descendant of *RealNumber.* When a resistor receives the message *resistance,* it looks up that property in its message dictionary. The *resistance* property is then evaluated in the context of the resistor, and simply returns itself. On the other hand, when a resistor receives the message *show,* its *show* property is looked up and evaluated in the context of the resistor. The code associated with the *show* property finds the set of point arrays and lines that constitute the resistor's image, and displays them on the screen.

Normally the ThingLab user will not describe an object by writing code, but rather will construct it incrementally by first making a descendant of a prototype, and then adding and editing properties. To start things off, there are a number of prototypical objects defined in the bare system. There are prototypes for mathematical objects such as real numbers, points, lines, and sets. Also, there are prototypes for objects that implement actions such as showing, printing, merging, editing, and making descendants. For example, the object *Printer* is a prototype for objects that can produce a printed representation of another object. The object *Thing* is the parent of all other objects. It has defaults for properties such as *show, print, merge, edit,* and *new,* ensuring that every object will have <u>some</u> way of showing itself, of printing itself, and so on.

Constraints

A constraint restricts the behavior of a set of objects, i.e., it restricts their responses to messages. A constraint is itself an object, with two of its properties being the expressions *error* and *tolerance.* When the absolute value of the error is less than the tolerance, the constraint is satisfied. A constraint is applied by making it a property of some other object For example, to construct a prototypical object *Horizontal Line,* the user would make a descendant of *Line,* and add as a property a constraint that the *y* values of the endpoints be equal. The error expression for this constraint would simply be *pointJ y - point! y,* and the tolerance would be 1 (raster point). Descendants of *Horizontal Line* inherit this constraint in the usual way.

Constraint satisfaction begins when the user adds a new constraint or edits an existing one. The system finds the set of constraints that might no longer be satisfied, sends messages to these constraints requesting methods for satisfying them, and chooses and sets up methods. The results of this planning are saved, since the same methods can often be used many times to satisfy a given set of constraints.

If possible, a one-pass method is used. Such a method orders the constraints and the objects to which they apply, so that each object in turn can be updated to satisfy its constraints, without the use of backtracking or successive approximations. For constraints involving conditionals, the method of assumed states is used with a simple depth-first tree search of the space of possible truth values of the conditions. Finally, the relaxation method is employed in satisfying circular constraints on real numbers.

The methods described above work nicely when the constraints can be satisfied in one pass. For example, when the user is moving a part of a line drawing with some constraints on it, the system can redraw the picture quickly enough to provide rapid feedback. However, relaxation and the method of assumed states as implemented are too slow for use with large problems. There are several steps which could be taken to improve the situation. One step would be to build into the system a much more powerful set of constraint satisfaction methods, such as symbolic manipulation routines and dependency-directed backtracking fsec for example staiiman & Sussman 1976]. Also, better use should be made of the modularity of the object-oriented representation. Consider a simulation of a complex circuit containing a linear amplifier. As far as the rest of the circuit is concerned, the amplifier should simply be an object with some input and output nodes, i.e., it should be a black box. Internally, the amplifier will have descriptions of all its parts and their constraints. However, when its input is within the linear region, the amplifier should use a simple constraint to find its output. Only when the input is not in this region should it make use of the detailed descriptions of its parts.

Merging

Suppose that a new object *C* is to be constructed by merging the objects *A* and *B.* The way in which an object merges with another is described by its *merge* property. Although idiosyncratic interpretations of merging are possible, at present nearly all objects use the *merge* property inherited from the primordial object *Thing.* Using this property, *C* would be constructed as follows. *A* can merge with *B* only if they both have the same code. The properties of *C* are formed recursively by merging the like-named properties from *A* and *B.* If a property name is found in only one of the original objects, then the corresponding property is simply copied. The set of parents of C is the union of the sets of parents of *A* and *B.* *C* has all the constraints that apply to either *A or B.* A few primitive objects have some additional interpretations of merging. For example, if *A* is a set, then *B* must also be a set, and the contents of the new set C is the union of the contents of *A* and *B.*

Merging is also used to specify connectivity. Following the methods used in Sketchpad, connectivity is represented by merged parts. For example, in an electrical circuit simulation, the terminals of components are descendants of *Node,* just as the endpoints of a line are descendants of *Point.* To connect one component to another, a node from one of the components is merged with a node from the other to form a new node. This new node is then substituted for the old ones.

The properties of the new node are formed by merging the like-named properties from the two original nodes. Two properties of nodes are *currents/n,* the set of currents flowing into the node, and *voltage,* the voltage at the node. The set of currents has a constraint that the sum of the currents be 0 (Kirchhoff's current law). The *currentsIn* property of the new node is formed by merging the *currents/n* properties from the original nodes, and contains all the currents from both of the original sets. The sum of its elements must still be 0. Similarly, the new *voltage* property has all the constraints applying to either of the original voltages. For example, if one of the components being connected was a resistor, the voltage at the resulting node would still be constrained by Ohm's law.

Current Status

The system is being written in Smalltalk-72. A version is running that implements all the features described here. So far, however, it has been tried only on small examples, and is still being actively expanded and modified.

Acknowledgements

Many people, both in LRG and elsewhere, have provided valuable advice on this project. In particular, I'd like to thank Danny Bobrow, Adele Goldberg, Dan Ingalls, Ted Kaehler. Alan Kay. Dave Robson, David Shaw. Rich Steiger. Steve Weyer, and Terry Winograd. Xerox Palo Alto Research Center has provided support and excellent facilities. This project is being done as a Ph.D. thesis in computer science at Stanford University.

References

Bobrow, Daniel G., and Winograd. Terry, "An Overview of KRL, A Knowledge Representation Language". *Cognitive Science,* V. 1, No. 1, 1977.

Dahl, Ole-Johan, Myhrhaug, Bjrirn, and Nygaard, Kristen, *Common Base Language,* Norwegian Computing Center Publication S-22, Oslo, Norway, October 1970.

Goldberg, Adele. and Kay. Alan (eds.). *Smalltalk-72 Instruction Manual,* Xerox Palo Alto Research Center, SSL 76-6, 1976.

Hewitt, Carl, *Viewing Control Structures as Patterns of Passing Messages,* MIT AI Lab Memo 410, Dec 1976.

Learning Research Group, *Personal Dynamic Media,* Xerox Palo Alto Research Center. SSL 76-1, 1976. Appears in part in Alan Kay and Adele Goldberg. "Personal Dynamic Media". *IEEE Computer,* March 1977, pp. 31-41.

Slallman, Richard M., and Sussman, Gerald J., *Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis,* MIT AI Lab Memo 380, Sept 1976.

Sutherland, Ivan E., *Sketchpad: A Man-Machine Graphical Communication Svstem,* Ph.D. thesis, MIT, Cambridge, Mass., 1963.