DESCRIPTIONS FOR A PROGRAMMING ENVIRONMENT

Ira P. Goldstein and Daniel G. Bobrow
Xerox Palo Alto Research Center
Palo Alto, California 94304, U.S.A

Abstract

PIE is an experimental personal information environment implemented in Smalltalk that uses a description language to support the interactive development of programs. PIE contains a network of nodes, each of which can be assigned several perspectives. Each perspective describes a different aspect of the program structure represented by the node, and provides specialized actions from that point of view. Contracts can be created that monitor nodes describing different parts of a program's description. Contractual agreements are expressible as formal constraints, or, to make the system failsoft, as English text interpretable by the user. Contexts and layers are used to represent alternative designs for programs described in the network. The layered network database also facilitates cooperative program design by a group, and coordinated, structured documentation.

Introduction

In most programming environments, there is support for the text editing of program specifications, and support for building the program in bits and pieces. However, there is usually no way of linking these interrelated descriptions into a single integrated structure. The English descriptions of the program, its rationale, general structure, and tradeoffs are second class citizens at best, kept in separate files, on scraps of paper next to the terminal, or, for a while, in the back of the implementor's head.

Furthermore, as the software evolves, there is no way of noting the history of changes, except in some primitive fashion, such as the history list of Interlisp [10]. A history list provides little support for recording the purpose of a change other than supplying a comment. But such comments are inadequate to describe the rationale for coordinated sets of changes that are part of some overall plan for modifying a system. Yet recording such rationales is necessary if a programmer is to be able to come to a system and understand the basis for its present form.

Developing programs involves the exploration of alternative designs. But most programming environments provide little support for switching between alternative designs or comparing their similarities and differences. They do not allow alternative definitions of procedures and data structures to exist simultaneously in the programming environment; nor do they provide a representation for the evolution of a particular set of definitions across time.

In this paper we argue that by making descriptions first class objects in a programming environment, one can make life easier for the programmer through the life cycle of a piece of software. Our argument is based on our experience with PIE, a description-based programming environment that supports the design, development, and documentation of Smalltalk programs.

Networks

The PIE environment is based on a network of nodes which describe different types of entities. We believe such networks provide a better basis for describing systems than files. Nodes provide a uniform way of describing entities of many sizes, from small pieces such as a single procedure to much larger conceptual entities. In our programming environment, nodes are used to describe code in individual methods, classes, categories of classes, and configurations of the system to do a particular job. Sharing structures between configurations is made natural and efficient by sharing regions of the network.

Nodes are also used to describe the specifications for different parts of the system. The programmer and designer work in the same environment, and the network links elements of the program to elements of the design and specification. The documentation on how to use the system is embedded in the network also. Using the network allows multiple views of the documentation. For example, a primer and a reference manual can share many of the same nodes while using different organizations suited to their different purposes.

In applying networks to the description of software, we are following a tradition of employing semantic networks for knowledge representation. Nodes in our network have the usual characteristics that we have come to expect in a representation language--for example, defaults, constraints, multiple perspectives, and context-sensitive value assignments.

There is one respect in which the representation machinery developed in PIE is novel: it is implemented in an object-oriented language. Most representation research has been done in Lisp. Two advantages derive from this change of soil. The first is that there is a smaller gap between the primitives of the representation language and the primitives of the implementation language. Objects are closer to nodes (frames, units) than lists. This simplifies the implementation and gains some advantages in space and time costs. The second is that the goal of representing software is simplified. Software is built of objects whose resemblance to frames makes them natural to describe in a frame-based knowledge representation.

Perspectives

Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. For example, one view of a Smalltalk class provides a definition of the structure of each instance, specifying the fields it must contain; another describes a hierarchical organization of the methods of the class; a third specifes various external methods called from the class; a fourth contains user documentation of the behavior of the class.

The attribute names of each perspective are local to the perspective. Originally, this was not the case. Perspectives accessed a common pool of attributes attached to the node. However, this conflicted with an important property that design environments should have, namely, that different agents can create perspectives independently. Since one agent cannot know the names chosen by another, we were led to make the name space of each perspective on a node independent.

Perspectives may provide partial views which are not necessarily independent. For example, the organization perspective that categorizes the methods of a class and the documentation perspective that describes the public messages of a class are interdependent. Attached procedures are used to maintain consistency between such perspectives.

Each perspective supplies a set of specialized actions appropriate to its point of view. For example, the *print* action of the structure perspective of a class knows how to prettyprint its fields and class variables, whereas the organization perspective knows how to prettyprint the methods of the class. These actions are implemented directly through messages understood by the Smalltalk classes defining the perspective.

Messages understood by perspectives represent one of the advantages obtained from developing a knowledge representation language within an object-oriented environment. In most knowledge representation languages, procedures can be attached to attributes. Messages constitute a generalization: they are attached to the perspective as a whole. Furthermore, the machinery of the object language allows these messages to be defined locally for the perspective. Lisp would insist on global functions names.

Contexts and Layers

All values of attributes of a perspective are relative to a *context*. Context as we use the term derives from Conniver [9]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is natural to create alternative contexts in which different values are stored for attributes in a number of nodes. The user can then examine these alternative designs, or compare them without leaving the design environment. Since there is an explicit model of the differences between contexts, PIE can highlight differences between designs. PIE also provides tools for the user to choose or create appropriate values for merging two designs.

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer. Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context the first time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Layers and contexts are themselves nodes in the network. Describing layers in the network allows the user to build a description of the rationale for the set of coordinated changes stored in the layer in the same fashion as he builds descriptions for any other node in the network. Contexts provide a way of grouping the incremental changes, and describing the rationale for the group as a whole. Describing contexts in the network also allows the layers of a context to themselves be asserted in a context sensitive fashion (since all descriptions in the network are context-sensitive). As a result, super-contexts can be created that act as *big switches* for altering designs by altering the layers of many sub-contexts.

Contracts and Constraints

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ contracts between nodes to describe these dependencies. Implementing contracts raises issues involving 1) the knowledge of which elements are dependent; 2) the way of specifying the agreement; 3) the method of enforcement of the agreement; 4) the time when the agreement is to be enforced.

PIE provides a number of different mechanisms for expressing and implementing contracts. At the implementation level, the user can attach a procedure to any attribute of a perspective, (see [2] for a fuller discussion of attached procedures); this allows change of one attribute to update corresponding values of others. At a higher level, one can write simple constraints in the description language (e.g. two attributes should always have identical values), specifying the dependent attributes. The system creates attached procedures that maintain the constraint.

There are constraints and contracts which cannot now be expressed in any formal language. Hence, we want to be able to express that a set of participants are interdependent, but not be required to give a formal predicate specifying the contract. PIE allows us to do this. Attached procedures are created for such contracts that notify the user if any of the participants change, but which do not take any action on their own to maintain consistency. Text can be attached to such informal contracts that is displayed to the user when the contract is triggered. This provides a useful inter-programmer means of communication and preserves a *failsoft* quality of the environment when formal descriptions are not available.

Ordinarily such non-formal contracts would be of little interest in artificial intelligence. They are, after all, outside the comprehension of a reasoning program. However, our thrust has been to build towards an artificially intelligent system through succcessive stages of man-machine symbiosis. This approach has the advantage that it allows us to observe human reasoning in the controlled setting of interacting with the system. Furthermore, it allows us to investigate a direction generally not taken in AI applications: namely the design of memory-support rather than reasoning-support systems.

An issue in contract maintenance is deciding when to allow a contract to interrupt the user or to propagate consistency modifications. We use the closure of a layer as the time when contracts are checked. The notion is that a layer is intended to contain a set of consistent values. While the user is working within a layer, the system is generally in an inconsistent state. Closing a layer is an operation that declares that the layer is complete. After contracts are checked, a closed layer is immutable. Subsequent changes must be made in new layers appended to the appropraiate contexts.

Coordinating designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment in which potentially overlapping partial designs can be examined without overwriting one another. This is accomplished by the convention that different designers place their contributions in separate layers. Thus, where an overlap occurred, the divergent values for some common attributes are in distinct layers.

Merging two designs is accomplished by creating a new layer into which are placed the desired values for attributes as selected from two or more competing contexts. For complex designs, the merge process is, of course, non-trivial. We do not, and indeed cannot, claim that PIE eliminates this complexity. What it does provides is a more finely grained descriptive structure than files in which to manipulate the pieces of the design. Layers created by a merger have associated descriptions in the network specifying the contexts participating in the merger and the basis for the merger.

Meta-description

Nodes can be assigned meta-nodes whose purpose is to describe defaults, constraints, and other information about their object node. Information in the meta-node is used to resolve ambiguities when a command is sent to a node having multiple perspectives.

One situation in which ambiguity frequently arises is when the PIE interface is employed by a user to browse through the network. When the user selects a node for inspection, the interface examines the meta-node to determine which information should be automatically displayed for the user. By appropriate use of meta-information, we have made the default display of the PIE browser identical to one used in Smalltalk. (Smalltalk code is organized into a simple four-level heirarchy, and the Smalltalk browser allows examination and modification of Smalltalk code using this taxonomy.) As a result, a novice PIE user finds the environment similar to the standard Smalltalk programming environment which he has already learned.

Simplifying the presentation and manipulation of the layered network underlying the PIE environment remains an important research goal, if the programming environment supported by PIE is to be useful as well as powerful. We have found use of a meta-level of descriptions to guide the presentation of the network to be a powerful device to achieve this utility.

Conclusion

PIE has been used to describe itself, and to aid in its own development. Specialized perspectives have been developed to aid in the description of Smalltalk code, and for PIE perspectives themselves. On-line documentation is integrated into the descriptive network. The implementors find this network-based approach to developing and documenting programs superior to the present Smalltalk programming environment. A small number of other people have begun to use the system.

This paper presents only a sketch of PIE from a single perspective. The PIE description language is the result of transplanting the ideas of KRL [2] and FRL [6] into the object oriented programming environment of Smalltalk [8], [7]. A more extensive discussion of the system in terms of the design process can be found in [1], and [4]. A view of the PIE description language as an extension of the object oriented programming metaphor can be found in [5]. Finally, the use of PIE as a prototype office information system is described in [3].

References

[1] Bobrow, D.G. and Goldstein, I.P. "Representing Design Alternatives", *Proceedings of the AISB Conference*, Amsterdam, 1980.

[2] Bobrow, D.G. and Winograd, T. "An overview of KRL, a knowledge representation language", *Cognitive Science* 1, 1 1977.

[3] Goldstein, I.P. "PIE: A network-based personal information environment", *Proceedings of the Office Semantics Workshop*, Chatham, Mass., June, 1980.

[4] Goldstein, I.P. and Bobrow, D.G., "A layered approach to software design", *Xerox Palo Alto Research Center CSL-80-5*. 1980a.

[5] Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference*. Stanford University, 1980b.

[6] Goldstein, I.P. and Roberts, R.B. "NUDGE, A knowledge-based scheduling program", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge: 1977, 257-263.

[7] Ingalls, Daniel H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp 9-16.

[8] Kay, A. and Goldberg, A. "Personal Dynamic Media" *IEEE Computer*, March, 1977.

[9] Sussman, G., & McDermott, D. "From PLANNER to CONNIVER -- A genetic approach". *Fall Joint Computer Conference*. Montvale, N. J.: AFIPS Press, 1972.

[10] Teitelman, W., *The Interlisp Manual*, Xerox Palo Alto Research Center, 1978.