

**Multiple Representation Perspectives
for Supporting Explanation in Context**

Christian Rathke and David F. Redmiles

CU-CS-645-93

March 1993

Christian Rathke

Institut fuer Informatik
Universitaet Stuttgart
Breitwiesenstrasse 20/22
W-7000 Stuttgart 80
Germany
phone: 011-45-711-7816-436
email: rathke@@informatik.uni-stuttgart.de

David Redmiles

Department of Computer Science
and Institute of Cognitive Science
University of Colorado
Campus Box 430
Boulder, Colorado 80309-0430
USA
phone: (303) 492-1503
email: redmiles@@cs.colorado.edu

Acknowledgements

The authors would like to thank Kerstin Drehmann, Gerhard Fischer, Egbert Lehmann, Jim Martin, Robert Rist, Petra Schmidt, Frank Shipman, and John Rieman for their helpful discussions and support. The research at C.U. was supported in part by the Army Research Institute under grant No@. MDA903-86-C0143 and a joint grant from the Colorado Advanced Software Institute (CASI) and US West Advanced Software Technologies Division.

CASI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado. CATI promotes advanced technology education and research at universities in Colorado for the purpose of economic development.

Multiple Representation Perspectives for Supporting Explanation in Context

Christian Rathke
Institut für Informatik
Universität Stuttgart

David F. Redmiles
Department of Computer Science
University of Colorado, Boulder

Abstract

The term *perspective* is used for a set of properties describing an object with respect to a common theme. The use of a perspectives representation is illustrated and evaluated in a programming tool called EXPLAINER, which supports programmers by providing explanations of examples related to the programmers' current task. Having multiple representation perspectives is essential for supporting people working from examples. Namely, it allows explanations of the examples to be generated according to perspectives that accommodate the changing context and needs of the programmers while they explore an example and develop a solution to their task. The multiple representation perspectives and corresponding perspective explanations support the development of a mental model of the example that can then be used in solving the current task.

A formal notion of perspectives as named sets of property-value pairs is incorporated in FrameTalk, a frame-based knowledge representation language. It is used to illustrate the representational basis for examples as they occur in the EXPLAINER system. The perspectives mechanism also addresses problems found in term-based knowledge representation languages, problems such as the proliferation of nonintuitive concepts, the misuse of the generalization link as a compositional link, and the differences when using concepts in different application contexts.

1 Introduction

A *perspective* is a way of viewing or interpreting an object, organizing various aspects according to various relations. The object may be concrete (e.g. a car, a person) or abstract (e.g. a mental plan how to get home on the bus). Which perspective is appropriate for interpreting an object depends on the context of a specific situation.

A common example of perspectives is in the reporting of current events. Hovy, for example, illustrates many different perspectives expressed in various texts describing a protest on the Yale campus [Hovy 88]. Though the events of the protest existed, in an absolute sense, as facts in the world, the different texts emphasized different aspects and implications. In this case, the perspectives varied based on the goals of the different narrators.

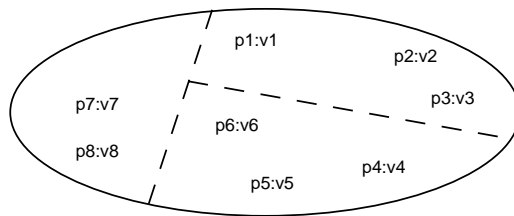
While many uses might be hypothesized for a formal scheme for multiple representation perspectives, a specific need arises in providing support for problem solving on the computer — programming. Namely, empirical studies have demonstrated that examples help people to learn and apply programming constructs [PirolliAnderson 85, KesslerAnderson 86, Lewis 88]. However, these same studies point to the conclusion that the users of the examples must be able to develop an appropriate *mental model* of the example in order to apply or transfer the relevant knowledge to new tasks. The work described below illustrates how multiple representation perspectives can be used to explain program examples for the purpose of helping programmers develop a mental model for solving new tasks. Thus, this work addresses both the problem of defining the form and behavior of multiple representation perspectives and the problem of applying that mechanism to support programmers in problem solving by analogy.

Section 2 provides a more formal description of a scheme for multiple representation perspectives and its relation to mental models and problem solving by analogy. Section 3 describes the application of these ideas in a tool called EXPLAINER. Section 4 motivates and details the representations needed by the EXPLAINER tool. Section 5 surveys some of the relevant literature. The concluding section discusses the general benefit of this approach to knowledge representation, especially with respect to knowledge-based human-computer applications.

2 Conceptual Framework

A *perspective* may be thought of as a set of properties for describing an object according to a particular, common theme. An object may be represented by several *instantiations*, each instantiation belonging to a distinct perspective and having specific values associated with that perspective's properties. For instance, the object in

Figure 1 may be represented by any of the three sets: $\{p_1 : v_1, p_2 : v_2, p_3 : v_3\}$, $\{p_4 : v_4, p_5 : v_5, p_6 : v_6\}$ and $\{p_7 : v_7, p_8 : v_8\}$. Perspectives may be related, i.e., property-value pairs of one perspective may depend on property-value pairs of other perspectives. Thus, a perspective defines a set of properties for describing an object, and an instantiation of the perspective associates concrete values to the properties. By *applying* a perspective to an object, the corresponding property set (the instantiation) is *selected* as the representative for the object.



An object is composed of three perspectives representing the object from three different points of view.

Figure 1: Perspectives of an Object

As with the news stories reported by Hovy, an absolute or whole object exists, but is seen only through instantiations of different perspectives. Any of the instantiations may serve as a representative for the overall object but it is unlikely that any instantiation would, in practice, include all properties.

The fragmentation of an object into constituent perspectives is also partly suggested by work in case-based reasoning [Schank 90, RiesbeckSchank 89]. In a sense, different cases or contexts in which an object is encountered and used could be interpreted as the source of specific perspectives. A particular aspect for understanding an object would be derived from and tightly coupled with a particular previous experience or situation (e.g., see [Alterman 88]). Using Figure 1, for instance, the set $L = \{p_7 : v_7, p_8 : v_8\}$ could represent a perspective on the whole object that was derived from case C_L .

This conceptualization of multiple representation perspectives provides a framework for understanding how examples can support a programmer while designing code for a programming task. Namely, if the object in Figure 1 represented a particular programmer's understanding of a programming task, then the programming process could break down if the programmer were missing a needed perspective, say L .

The breakdown could be repaired if the programmer were supplied with case C_L , recognized the missing perspective, and then applied it to the current task.

This framework is based on a comprehension-oriented theory in problem solving [KintschGreeno 85, DijkKintsch 83]. The programmer's understanding of a task is referred to as a *mental model*. In programming, the mental model of a task has been defined as a person's informal understanding of a task and the knowledge to transform that informal understanding into a formal model, e.g., a program code [Fischer 87]. Referring again to Figure 1, some of the perspectives could represent general knowledge about the task; some, the associations to previous situations; and some, the strategies for transforming elements of the task into the structure of a program code.

The strategy for transforming the task into a solution is a kind of knowledge usually termed a *programming plan* [Soloway et al. 88, SolowayEhrlich 84]. Programming plans are treated in analyses of programming as a single kind of knowledge. However, they actually interrelate knowledge from many perspectives. Specifically, the transformation of task information into a solution indicates a mapping between at least two perspectives, the problem domain of the task and the formalism of the solution (usually a programming language).

Thus, in the EXPLAINER tool illustrated below, a programming example is represented by the many perspectives and interrelationships that form the programming plan for the example. The translation of elements of a task into elements of a program solution is captured. Explanation consists of various techniques to visualize the different perspective instantiations and interrelationships. In a sense, the representation of the example tries to capture the many perspectives of the mental model the author of the example had when originally working out the example task.

The goal is to provide the example to a programmer when the programmer is trying to solve an analogous task. By seeing in the example, instantiations of the perspectives representing the programming plan solving the example task, the programmer then would be able to recognize perspectives that when applied to the current task would lead to a solution. For instance, if the set L is missing from the programmer's mental model, then it can be discovered in and retrieved from case C_L . Figure 1 is both an abstraction of the mental model of a programmer trying to complete a task, and an abstraction of the representation for an example.

The goal of this conceptual framework is clearly supported by the direction of the

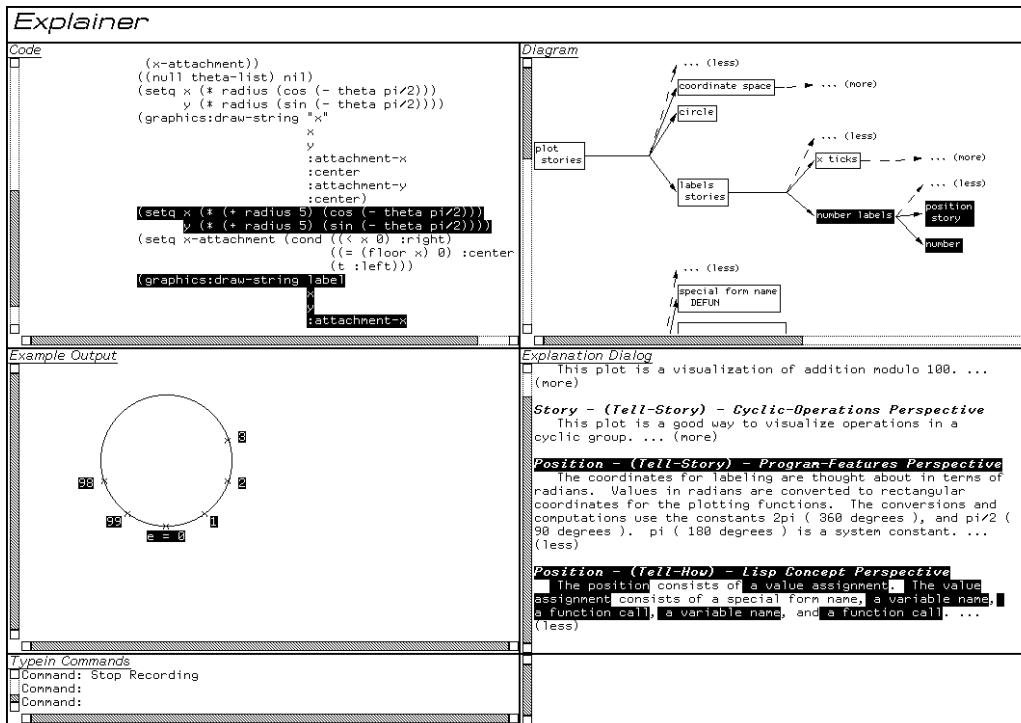
above mentioned research in case-based reasoning. It is also substantiated by empirical, cognitive studies of programmers successfully learning from and using examples by analogy, as noted in the introduction. Also noted was the issue that although examples helped the subjects in learning and problem solving, performance was dependent on the subjects developing the appropriate mental model. The goal of the EXPLAINER interface is to make the perspectives of the mental model obvious and accessible to programmers, especially the perspectives constituting the programming plan.

3 Perspectives, Views and Explanation

As described in the conceptual framework, the purpose of the EXPLAINER tool is to make apparent to the programmer, the programming plan through its many perspectives. In this exploratory work, the problem of providing the right perspective has been simplified by restricting the domain of the tasks and examples to graphics functions. The graphical features of an example serve as an entry point by which the programmer can compare the example to the requirements of the new task. In this sense, the existence of a common perspective between example and task is guaranteed; minimally, it is found in the graphics features of an example. If programmers have enough background knowledge to identify the graphical features they want, they can use the EXPLAINER interface to explore how features are mapped onto programming concepts. They can study the programming plan and apply some of its aspects to solving the current task.

The EXPLAINER interface presents *multiple views* of the information comprising an example: code listing, sample execution, component diagrams, and text (Figure 2). For instance, in the example of a LISP program for visualizing operations of the mathematical concept of a cyclic group, a LISP concept such as the call to the function that draws the circle might be presented as a stylized fragment of the code listing, as a circle graphic object in the sample execution, or as a node in a component diagram. The possibility to view the sample information in different external forms accommodates different persons' preferences. A person preferring text over diagrams would be able to access the same information. Redundant views also provide reinforcement of new concepts.

Within a view, the programmer can access information from *different perspectives*. Currently, the different perspectives are selected from a list (Figure 3b), appearing

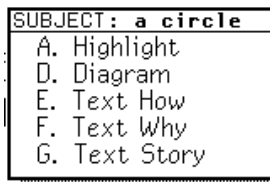


The above screen shows the actual state of the EXPLAINER interface at the end of one programmer's test session. The programmer was exploring the information in this example in learning how to complete the programming task of drawing a clock face (see Figure 4). The graphical analogy has led the subject to explore the example of a LISP program for visualizing operations of the mathematical concept of a cyclic group.

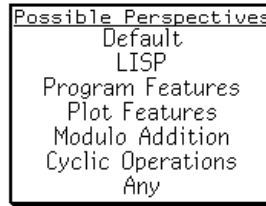
Figure 2: Exploring an Example in EXPLAINER

after a command is selected (Figure 3a). For instance, in the diagram view, the programmer has created diagrams of the example from the GRAPHICS and LISP perspectives (upper right of Figure 2—the LISP perspective being only partially visible). Text has been presented from LISP, PROGRAM-FEATURES, and CYCLIC-OPERATIONS perspectives (lower right of Figure 2).

Which perspectives are needed depend on the context of the current task or subtask and the needs of the programmer. For instance, in solving the programming task of Figure 4, the programmer might initially like to know that the example in Figure 2 had a feature such as circle drawing, i.e., need a GRAPHICS perspective. Later, the programmer might need to know what arguments the circle drawing function required, i.e., need a LISP or programming language perspective. Alternatively, the programmer might already be familiar with programming the circle and just explore



(a)



(b)

Figure 3: Pop-up Menus in EXPLAINER

the example to understand programming the labels.

Clock Programming Task

Write a program to draw a clock face (without the hands) that shows the hour numerals, 1 - 12. Your solution should look like the picture to the right.

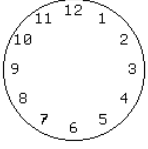


Figure 4: Clock Task as Described to Programmers

Thus, the EXPLAINER interface allows programmers to *access information about the programming plan from different perspectives and through different views*. Highlighting and textual descriptions allow programmers to understand the relationships between plan elements and system components. Figure 2 is the actual state of the EXPLAINER interface as left by one of the test subjects during the evaluation. The subject was exploring this example program to perform the programming task of Figure 4. The screen shows that the subject had

- expanded the diagram view to explore the components of labels in the plot-features perspective (clicking on “more” cues);
- redrawn the initial diagram from its plot-features perspective to a LISP perspective – only a portion is visible in the middle of the diagram pane (menu action “Diagram”);
- retrieved the story, displayed in the explanation-dialog view, about the concept of labels in the program-features perspectives (menu action “Text Story”);

- generated a description of how labels are implemented in the LISP perspective (menu action “How”); and
- highlighted the concepts having to do with labels common across the several different perspectives and the four views.

This specific information enabled the test programmer to identify the LISP function called to draw the label, the assignment function that calculated the position, and what variables the position calculation depended on. The programmer could then apply the same problem decomposition, program structure, and functions in the solution of the clock task, or in this case, simply modify a copy of the example to place the labels inside the perimeter of the circle.

In some cases, the task solved by the example might match closely the current task. In such cases, the mapping from the task perspective to the graphics feature perspective can support programmers. For instance, if instead of the clock task, programmers were given the task to illustrate arithmetic modulo 10, they could expand descriptions from the modulo arithmetic perspective and simply identify the mapping from there to the programming level. The study of examples where task descriptions match closely as compared to nonmatching tasks (e.g., the cyclic group example as compared to the clock task) is an area of future research.

The EXPLAINER tool is only one component of an example-based design process: Examples residing in a catalog repository must be retrieved by users. Retrieval requires users to articulate some specification of requirements in order for an appropriate example to be delivered. Once examples are retrieved, users must be supported in understanding the example in order to adapt it to their new task. Working with an example can lead to additional ideas about the problem requirement, possibly leading to refinement of the original requirements specification and retrieval of additional examples. Work supporting the requirements specification and the modification of examples within a design life cycle is reported elsewhere [Fischer et al. 92, Redmiles 92a].

Also, the study alluded to above is documented in [Redmiles 92a] and [Redmiles 92b]. In this study of 25 test programmers, it was found that overall, the EXPLAINER users were successful in accessing the information of the programming plan for the example and applying it to a new task. Furthermore, the variability of performance

by EXPLAINER users was reduced as a group, implying that programmers within the group who needed more help (e.g., had poorer mental models with respect to solving the task) were in fact helped.

4 Representation

For illustrating our approach to representation, we will use the concrete syntax of FrameTalk [Rathke 91], a frame language with descriptive elements for defining abstract frames, perspectives, slots, slot restrictions and attributes. These elements are used to define the terminological level of the representation. FrameTalk's assertional level consists of structured objects comprised of instantiations with properties and values, which can be used to perform various reasoning processes typically associated with term-based knowledge representation languages such as realization and propagation.

In general, knowledge representation schemes provide only limited support for specifying the domain concepts of an application. The primitives of a knowledge representation language only intuitively suggest how they may be used to formulate knowledge about a domain. It is a major task of the knowledge engineer to find the "right way" of describing application objects in a conceptual taxonomy. For instance, there is often a tendency to define concepts for combinations of properties. Concepts such as VOLLEY-BALL-PLAYER, FOOT-BALL-PLAYER, TEACHER and MANAGER are combined (as in VOLLEY-BALL-PLAYING-TEACHER, FOOT-BALL-PLAYING-TEACHER, VOLLEY-BALL-PLAYING-MANAGER etc.) to describe a person fitting the common description of the two or more concepts. The new concept is made a *sub-concept* of the existing ones and thereby *inherits* all what is specific for them. This leads to some unwanted consequences for an application taxonomy:

- There is a combinatorial proliferation of concepts.
- Concepts tend to become artificial; they do not have a natural counterpart in the conceptual structure of the application domain.
- The taxonomic relationships between concepts do not allow to distinguish between different dimensions of generalization. TEACHER is a more general concept than VOLLEY-BALL-PLAYING-TEACHER with respect to some sporting

activity. VOLLEY-BALL-PLAYER is more general than VOLLEY-BALL-PLAYING-TEACHER with respect to properties which characterize professions. The network of concepts represents multiple taxonomies which can not be separated based on its structure.

A related problem manifests itself when the generalization hierarchy with multiple super concepts is misused to combine partial definitions. In the area of object-oriented programming, these partial definitions are known as *mixin classes*. A super class link to a mixin class does not express generalization but composition of structure and behavior.

The need to compose descriptions from various sources is often caused by multiple usage contexts. In different application contexts, concepts are *used* in different ways. For a travel agent, a “person viewed as a traveler” (compare [BobrowWinograd 77]) looks different than a “person viewed as a dog-owner”. These differences in use are not expressed by the concept taxonomy.

The perspectives mechanism is designed to address some of these problems and provide additional structural support for representing knowledge about an application domain such as the one of the EXPLAINER system. The development of the perspectives mechanism for FrameTalk was guided by the following requirements:

1. It should be possible to *define* concepts or terms through a composition of perspectives, e.g., a person has properties related to his physical existence, his family and social context, his working context, etc.
2. It should be possible to *view* objects from different perspectives and to *apply* new perspectives to an object, e.g., depending on the context it should be possible to view a person as a student, employee, musician, etc.
3. There should be no structural difference between the definition and the view of a concept, i.e., any perspective which is used for defining a concept must also be usable as a point of view. The difference should be one of *usage* not one of structure.

The terminological level of FrameTalk allows perspectives to be specified. Technically, perspectives are named subsets of a frame’s slots. Figure 5 shows how the three perspectives for the cyclic group example shown in Figure 2 are specified.

```

(defframe cyclic-group (group)
  (:perspectives
    (group-theory (operator (:one function) (:diff components))
                  (elements (:all number) (:diff components)))
    (lisp (function (:one function)
                   (:diff components))
          (arguments (:all lisp-variable) (:diff components))
          (body (:all lisp-form) (:diff components)))
    (graphics (coordinate-space (:diff components))
              (circle (:diff components))
              (labels (:all graph-label) (:diff components))))
  (:relations
    (same-object (group-theory operator) (graphics circle))
    (same-object (group-theory elements)
                 (graphics labels number-labels)))

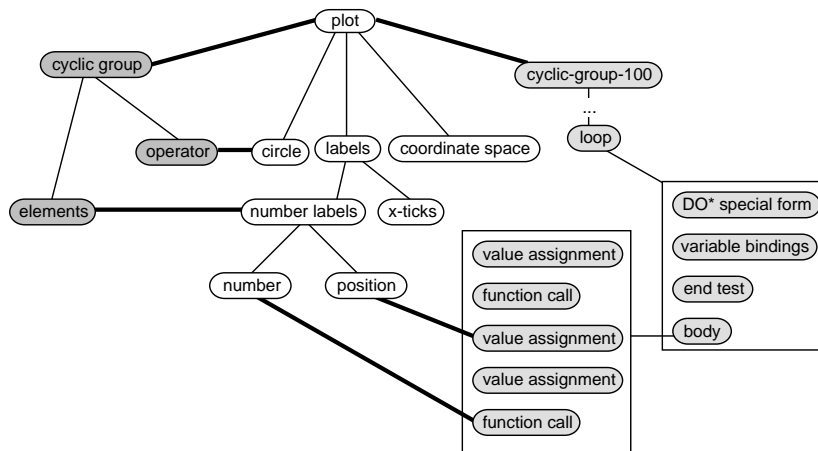
```

Figure 5: FrameTalk Definition of the Cyclic-Group Example

The cyclic-group example is represented by three perspectives: GROUP-THEORY, LISP and GRAPHICS. Each of the perspectives consists of slots with slot descriptions, which restrict possible slot fillers. For instance, the operator slot of the GROUP-THEORY perspective must be filled with exactly one function and all of the values of the elements slot must be numbers. In this example, all slots mentioned *differentiate* a components slot, which is defined at a more general level (not shown) in each of the three perspectives. Fillers of a component slot belong to the same perspective as the owner of the slot.

The relations part of the frame definition specifies the *same-object* relation for components in different perspectives. This relation ensures that slot fillers point to the same object (though potentially in different perspectives).

The instantiations which correspond to the cyclic group example form a network of nodes which are connected by three kinds of links: *components*, *roles* and *perspectives* (Figure 6). The components links are specialized to domain specific links such as operator, elements, function, They connect one instantiation to zero or more instantiations which comprise it *in the same perspective*. In the above example, the graphics perspective of the cyclic-group frame consists of a coordinate-space, a circle, and labels.



Three perspectives are shown in this figure: GRAPHICS (unshaded ovals), LISP (shaded ovals), and GROUP-THEORY (darkest ovals). The thin lines represent components/roles links. The thick lines correspond to perspectives links.

Figure 6: Partial Representation of an Example

The components link captures the “how to” or implementation knowledge in an example. The roles link (which is not explicitly mentioned in the definition) is the inverse of the components link and supplies one kind of “why” or goal knowledge. For example, a “circle” is drawn as part of the graphics for a cyclic group. Instantiations exist in one specific perspective. However, they can have equivalent or analogous counterparts in other perspectives. For example the “operator” and “elements” components of the group theory perspective are equivalent to the “circle” and “number-labels” components of the graphics perspective.

The frame definition of Figure 5 is an abstract description of the cyclic group example. The network of Figure 6 is subsumed by this abstraction. Some of the components/roles links and some of the perspectives links are specific for this example. All of them have been explicitly asserted by the programmer, for instance, the perspectives links between the position of the number labels in the graphics perspective and the value assignment form in the the LISP perspective. They explicitly state that the LISP form corresponds to the positioning of the graphical elements.

The abstraction could also be used to enforce perspective relations on instantiations. Asserting the fact that a given LISP program is described by the LISP perspective of the cyclic group frame generates instantiations for the other perspectives and establishes the necessary relationships. The remaining details, which are specific to the example,

would be filled in by the programmer.

In sum, instantiations in different perspectives are composed into hierarchies along the components/role relation. The hierarchies in different perspectives are interrelated through the perspectives links of individual nodes. Thus, as a whole, an example program is one structured network that may be interpreted according to various perspectives.

From a formal point of view, perspectives are represented by sets of slots with slot restrictions. Frames are composed of *descriptions* each of which belongs to a distinct perspective. Due to the perspectives structure of frames, an object is represented as a composition of instantiated descriptions. An object may be accessed by any of its parts, i.e., through any of its perspectives. Reasoning components of FrameTalk make use of this representation. For example, the realizer, a component which associates abstract descriptions to instantiations, operates within perspectives. If it finds a more specific description by which an instantiation is subsumed, it establishes a specific link between the instantiation and the description.

The reasoning done by the EXPLAINER system serves the intended purpose of presenting information about an example. Using the “Highlight” command (Figure 3a) initiates a search through the network for instantiations which are related by perspective links. At the least, instantiations are related through the *root* instantiations which is equivalenced to the main function of the LISP code for that example. Thus, any instantiation can be reached from any other, though some are distant with respect to the number of intervening links and nodes.

The search begins with a concept that has been presented in one of the views in the EXPLAINER interface. It traverses the links connecting nodes in the network (Figure 6) and ends with a related node or nodes. The target is often immediate nodes on the components link to respond to a “how” request or the immediate nodes on the perspectives link to fulfill a highlighted request. However, the search can traverse both components and role links any number of times, although cycles terminate the search in failure. The search pattern is basically breadth-first, though various characteristics of the path can be controlled to find the nodes in specific perspectives, views, or distances.

5 Related Work

Many knowledge-based systems exist or are being developed to support programmers and software designers. In general the distinctions could be summarized according to where emphasis is placed. Our emphasis while developing a knowledge-based tool has been on the design process as tightly coupled with a human designer and not on automation. A balance is sought between the creative abilities of the human and the ability to retrieve plans and examples by the computer. The assumption is that good design is still the domain of people, though systems can provide support [Brooks Jr. 87]. Systems supporting human designers have often focussed on only one perspective, the system/programming language perspective [RichWaters 90, TeitelmanMasinter 84]. Some systems support decomposition from domain perspectives, though the knowledge is often organized around a domain taxonomy instead of human-centered needs [Devanbu et al. 91, HarrisJohnson 91].

In the conceptual framework section, it was noted that the example-based approach was in part based on a comprehension oriented theory in problem solving explored by Kintsch and Greeno [KintschGreeno 85, DijkKintsch 83]. Their original work was in the area of word arithmetic problems and later was extended with Mannes to the domain of simulating computing tasks [MannesKintsch 91]. Their approach is to simulate how people use what knowledge in performing these tasks. Problem solving interpreted in a comprehension framework is particularly well-suited for understanding the role of example-based support as there is a dual role for comprehension: comprehension of the problem or task, and comprehension of a supplied example related to the task. In adapting parts of their framework, the emphasis had to be added with respect to how examples would have to be explored by people from the different perspectives specifically involved in programming plans. The different perspectives additionally provide a finer-grained analysis than their two-part separation of a person's understanding into situation and problem (mental) models.

Minsky [Minsky 75] introduced the notion of "frame-systems" for explaining a human's efficiency when interpreting visual scenes. Although movements around an object such as a cube changes the perceived image, there is no need to re-interpret the entire image because multiple views are represented by frames which share information in common "terminals". The idea of sharing parts is extended to apply to different frame-systems, i.e., collections of frames describing structurally different

objects. As an example, an electricity generator can be described as a mechanical and as an electrical system. Although the term “perspective” is not explicitly used, the resemblance to our approach is obvious.

The notion of perspectives has been used in the areas of knowledge representation and object-oriented programming. In MERLIN [MooreNewell 73] perspectives are mappings between so-called β -structures. A perspective is an alternative way of expressing the same thing. [+ 3 5], [8], [ARITH-OP BINARY COMMUTATIVE [FUNCTION ADDITION] [FIRST-ARG 3] [SECOND-ARG 5]] are alternative descriptions for the same object.

This notion of a perspective differs from the approach taken by KRL [BobrowWino-grad 77]. KRL descriptions are partial characterizations which may be applicable to an object. Descriptions can be compared to determine whether they are compatible, i.e. whether they can describe the same object. In this sense, descriptions may be used to express different point of views. Although one of the descriptor types of KRL is named “perspective” it is not directly related to multiple views. It is used to relate a description to a prototype which resembles the instance-class relationship of an object-oriented programming language.

In both MERLIN and KRL, the idea of multiple descriptions for the same entity are similar with our approach. Different expressions refer to the same entity and can be used as its representative. The different forms focus the inference processes by emphasizing important properties.

6 Conclusions

The proposed perspectives mechanism extends representation languages by providing additional structure for formulating and using knowledge in a taxonomy of concepts:

- Perspectives allow the formulation of *partial definitions*. Composition of properties can be structurally separated from specialization of properties.
- Different views can be expressed by combining properties in different ways. Objects may be described from different viewpoints by multiple perspectives.
- The perspectives structure can be used for comparing objects and answering questions for explanation purposes.

Furthermore, the multiple representation perspectives serves the dual role of providing a framework for organizing knowledge according to the needs of the mental models people rely upon during problem solving. This supports better provision and structuring of the knowledge represented for explanation.

The duality of perspectives for representation and perspectives for modeling the needs of human users is an elegant combination for a human-computer interface. It suggests an issue for the future: i.e., what properties a knowledge representation scheme for a human-computer application should have in order to promote the most natural balance and operation between computer representation of the knowledge and the human use of that knowledge. We believe this paper points to some of the desired characteristics.

Acknowledgements

The authors would like to thank Kerstin Drehmann, Gerhard Fischer, Egbert Lehmann, Jim Martin, Robert Rist, Petra Schmidt, Frank Shipman, and John Rieman for their helpful discussions and support. The research at C.U. was supported in part by the Army Research Institute under grant No. MDA903-86-C0143 and a joint grant from the Colorado Advanced Software Institute and US West Advanced Software Technologies Division.

References

- [Alterman 88] R. Alterman. Adaptive Planning. *Cognitive Science*, 12(3):393–421, 1988.
- [BobrowWinograd 77] D. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.
- [Brooks Jr. 87] F. Brooks Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.
- [Devanbu et al. 91] P. Devanbu, R. Brachman, P. Sefriddle and B. Ballard. LaSSIE: A Knowledge-Based Software Information System. *Communications of the ACM*, 34(5):34–49, 1991.

- [DijkKintsch 83] T. van Dijk and W. Kintsch. *Strategies of Discourse Comprehension*. Academic Press, New York, 1983.
- [Fischer et al. 92] G. Fischer, A. Girgensohn, K. Nakakoji and D. Redmiles. Supporting Software Designers with Integrated, Domain-Oriented Design Environments. *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Engineering*, 18(6):511–522, 1992.
- [Fischer 87] G. Fischer. Cognitive View of Reuse and Redesign. *IEEE Software, Special Issue on Reusability*, 4(4):60–72, July 1987.
- [HarrisJohnson 91] D. Harris and W. Johnson. Sharing and Reuse of Requirements Knowledge. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering (KBSE-91) Conference (Syracuse, NY)*, pp. 65–77, New York, September 1991. Rome Laboratory.
- [Hovy 88] E. Hovy. *Generating Natural Language Under Pragmatic Constraints*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.
- [KesslerAnderson 86] C. Kessler and J. Anderson. Learning Flow of Control: Recursive and Iterative Procedures. *Human-Computer Interaction*, 2:135–166, 1986.
- [KintschGreeno 85] W. Kintsch and J. Greeno. Understanding and Solving Word Arithmetic Problems. *Psychological Review*, 92:109–129, 1985.
- [Lewis 88] C. Lewis. Why and How to Learn Why: Analysis-Based Generalization of Procedures. *Cognitive Science*, 12(2):211–256, 1988.
- [MannesKintsch 91] S. Mannes and W. Kintsch. Routine Computing Tasks: Planning as Understanding. *Cognitive Science*, 3(15):305–342, 1991. also published as Technical Report No. 89-8, Institute of Cognitive Science, University of Colorado, Boulder, CO.
- [Minsky 75] M. Minsky. A Framework for Representing Knowledge. In P. Winston (Ed.), *The Psychology of Computer Vision*, pp. 211–277. McGraw Hill, New York, 1975.

- [MooreNewell 73] J. Moore and A. Newell. How can MERLIN understand? In L. Gregg (Ed.), *Knowledge and Cognition*, pp. 201–310. Lawrence Erlbaum Associates, Hillsdale, NJ, 1973.
- [PirolliAnderson 85] P. Pirolli and J. Anderson. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology*, 39(2):240–272, 1985.
- [Rathke 91] C. Rathke. Implementing Frames in an Object-oriented Programming Language. In A. Mrazik (Ed.), *Proceedings of the First East European Conference on Object-Oriented Programming*, pp. 88–94, Bratislava, CSFR, September 1991.
- [Redmiles 92a] D. F. Redmiles. *From Programming Tasks to Solutions – Bridging the Gap Through the Explanation of Examples*. PhD thesis, Department of Computer Science, University of Colorado, Boulder, CO, 1992.
- [Redmiles 92b] D. F. Redmiles. Reducing the Variability of Programmers’ Performance Through Explained Examples. in preparation, 1992.
- [RichWaters 90] C. Rich and R. Waters. *The Programmer’s Apprentice*. Addison-Wesley Publishing Company, Reading, MA, 1990.
- [RiesbeckSchank 89] C. Riesbeck and R. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.
- [Schank 90] R. Schank. *Tell Me a Story: A New Look at Real and Artificial Memory*. Charles Scribner’s Sons, New York, 1990.
- [Soloway et al. 88] E. Soloway, J. Pinto, S. Letovsky, D. Littman and R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.
- [SolowayEhrlich 84] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5):595–609, September 1984.
- [TeitelmanMasinter 84] W. Teitelman and L. Masinter. *The Interlisp Programming Environment*, pp. 83–96. McGraw-Hill Book Company, New York, 1984.