A Self-Watching Model of Analogy-Making and Perception

James B. Marshall
Department of Computer Science
Sarah Lawrence College
Bronxville, New York 10708
jmarshall@slc.edu

Abstract

This paper describes Metacat, an extension of the Copycat model of analogy-making. The development of Copycat focused on modeling context-sensitive concepts and the ways in which they interact with perception within an abstract microworld of analogy problems. This approach differs from most other models of analogy in its insistence that concepts acquire their semantics from within the system itself, through perception, rather than being imposed from the outside. The present work extends these ideas by incorporating self-perception, episodic memory, and reminding into the model. These mechanisms enable Metacat to explain the similarities and differences that it perceives between analogies, and to monitor and respond to patterns that occur in its own behavior as it works on analogy problems. This introspective capacity overcomes several limitations inherent in the earlier model, and affords the program a powerful degree of self-control. Metacat's architecture includes aspects of both symbolic and connectionist systems. The paper outlines the principal components of the architecture, analyzes several sample runs and examples of program-generated commentary about analogies, and discusses Metacat's relation to some other well-known models of analogy.

1 Introduction

This paper describes a computational model of analogy-making and perception called Metacat, which is based on the earlier Copycat model developed by Hofstadter and Mitchell (Hofstadter, 1984; Mitchell, 1993). Like Copycat, Metacat models the complex interplay of bottom-up and top-down processes involved in perception, using an emergent architecture that incorporates aspects of both symbolic and connectionist systems. Metacat, however, builds on the earlier model by focusing on *self*-perception and its relation to other cognitive processes. The long-term goal of this line of research is to understand how high-level cognitive phenomena such as concepts, analogical thinking, creativity, and self-awareness can emerge from a subcognitive substrate composed of a large number of fine-grained, nondeterministic actions, each of which is far too small by itself to support such phenomena.

Few people would claim that the individual neurons making up a human brain are "conscious" in anything like the normal sense in which humans experience consciousness. We are forced to accept the fact that self-awareness arises, somehow, out of nothing but billions of low-level chemical reactions and neuronal firings. How can individually meaningless physical events in a brain—even a huge number of them—ultimately give rise to meaningful awareness and understanding? Hofstadter has argued that two ideas are of paramount importance:

> What seems to make brains conscious is *the special way they are organized*—in particular, the higher-level structures and mechanisms that come into being. I see two dimensions as being critical: (1) the fact that brains possess *concepts*, allowing complex representational structures to be built that automatically come with associative links to all sorts of prior experiences, and (2) the fact that brains can *self-monitor*, allowing a complex internal self-model to arise, allowing the system an enormous degree of self-control and open-endedness. (Hofstadter & FARG, 1995)

The development of Copycat was intended to explore the first idea, by creating a computer model of analogy-making in which the representation of concepts is deeply intertwined with the program's mechanisms for high-level perceptual processing. Concepts in Copycat are not modeled as static representational structures; rather, they are dynamic entities that respond to perceptual processing in a highly context-sensitive way, bending and adapting to the situation at hand in a flexible manner. Furthermore, they actively influence perceptual processing itself. This tight coupling of concepts and perception in the model gives rise to an ability to perceive similarities between different situations by describing them in terms of a common set of underlying concepts applicable to both situations. The ability of Copycat to make analogies is a direct consequence of the nature of the program's representation of concepts.

The Metacat model explores the second idea, by endowing Copycat with a capacity for *self-watching*, defined here as the ability of a system to perceive—and to create explicit representations of—its own perceptual processes. Our objective has been to develop mechanisms that allow the program to monitor its own activity and to explicitly characterize the conceptual associations that implicitly arise as it solves analogy problems (Marshall & Hofstadter, 1997; Marshall, 1999). This can be thought of as adding a higher "cognitive" layer on top of Copycat's "subcognitive" layer, enabling the program to watch and remember what happens at its subcognitive level as perceptual structures are built, reconfigured, and destroyed. This type of self-reflective awareness is common in humans, who are quite capable of paying attention to, and explicitly articulating, patterns in their own thinking (Chi, Bassok, Lewis, Reimann, & Glaser, 1989; Chi, de Leeuw, Chiu, & LaVancher, 1994).

Copycat and Metacat are concerned with *high-level* perception, by which we mean that level of perceptual processing in which *concepts* play a critical role (Chalmers, French, & Hofstadter, 1992). In contrast, *low-level* perception refers to the processing of raw, modality-specific sensory data obtained directly from the environment, such as the detection of edges in retinal images, or the processing of audio frequencies from the inner ear, without regard to the meaning of this information. Low-level perceptual processing is the first step along the path leading to high-level perception, with many intermediate processing stages lying in-between involving ever greater degrees of abstraction. The end result of this process is the conscious recognition or understanding of the input stimulus as an instance of a particular mental concept or set of concepts.

Consider, for example, the everyday experience of recognizing your mother. A pattern of light falls on the hundred million or so photoreceptor cells in your retina, and a fraction of a second later, the idea of your mother comes to mind. A particular mental concept has become highly activated, while most others remain dormant. This process of recognition, for the most part, takes place below the level of conscious awareness. One does not have to do much deliberate thinking in order to recognize one's mother (at least in the absence of degraded environmental conditions such as poor lighting). High-level perception depends largely on *subcognitive* processing mechanisms (Hofstadter, 1985b).

The activation of the concept of *mother* elicited by a facial image is a relatively simple example of high-level perception in action. This same phenomenon, however, often occurs in more abstract contexts, such as when a person hears an unfamiliar piece of music for the first time and recognizes it as coming from a particular musical period or composer, or when a painting is recognized to be, say, an Impressionist work, or as belonging to Picasso's "Blue period". Moving to an even higher level of abstraction, a complicated social situation involving tangled webs of people, objects, relationships, and conflicting choices may collectively be perceived as a "Catch-22" situation. Even the concept of *mother* is, in reality, a subtle matter. Depending on context, a wide variety of things can be viewed as abstract instances of this concept. The Earth, for example, is sometimes described as the mother of all living things, an idea commonly expressed by the phrase “Mother Earth”. Strictly speaking, of course, considering a planet to be a mother makes no sense, but given the right context we can effortlessly see how the idea applies, thanks to the natural flexibility of human concepts.

In general, concepts in the mind are not sharply-defined entities with clear-cut boundaries, always applying to certain things but never to others. Rather, the boundaries of concepts are inherently ill-defined and blurry, and are strongly influenced by the context in which perception occurs. We refer to this type of inherent flexibility as *conceptual fluidity*, in order to stress the idea of concepts as nonrigid, adaptable, and highly context-sensitive. Much work has been done in cognitive psychology investigating the nature of the distances between concepts and categories (see, for example, Tversky, 1977; Smith & Medin, 1981; Goldstone, Medin, & Gentner, 1991). In particular, the strength of associations between concepts can change dynamically, according to context. Under the right pressures, concepts that are normally far apart may be brought close together, so that they are both seen as applying to a particular situation (such as when the Earth is regarded simultaneously as an instance of *planet* and *mother*). This phenomenon, which we refer to as *conceptual slippage*, is what enables apparently dissimilar situations to be perceived as being “the same” at a deeper, more abstract level.

Copycat and Metacat differ in important ways from many other models of analogy proposed by researchers in AI and cognitive science. See French (2002) for a recent overview. Probably the most important difference is the emphasis our models place on the representation of concepts, and the role played by concepts in making analogies. Other well-known models have focused on the mechanisms and psychological constraints involved in mapping a source situation to a target situation (Gentner, 1983; Falkenhainer, Forbus, & Gentner, 1990; Forbus, Ferguson, & Gentner, 1994); on the satisfaction of multiple competing constraints when constructing this mapping (Holyoak & Thagard, 1989); on the mechanisms that allow stored analogs to be retrieved from memory (Forbus, Gentner, & Law, 1995; Thagard, Holyoak, Nelson, & Gochfield, 1990; Kolodner, 1993); on the integration and mutual interaction of processes responsible for retrieval, mapping, and transfer

(Kokinov & Petrov, 2001; Eskridge, 1994); and on distributed representations of structure (Hummel & Holyoak, 1996, 1997; Holyoak & Hummel, 2001; Halford, Wilson, Guo, Gayler, Wiles, & Stewart, 1994; Wilson, Halford, Gray, & Phillips, 2001). All of these issues are important, and any full and satisfying theory of analogy should certainly include an account of them. In our view, however, a complete theory must also integrate concepts, perception, and meaning into the picture.

When humans make analogies, we not only construct mental mappings according to constraints, we also understand the *meaning* of the concepts connected by these mapping-structures. For example, a person making an analogy between a situation involving water and another involving heat presumably maps mental structures representing water to structures representing heat, at some level of abstraction. But people also understand what the underlying concepts of water and heat *mean*, from long experience with these concepts in the world. Of course, the act of making the analogy deepens this understanding by facilitating a transfer of knowledge from one situation to the other. But the important point is that the constituent concepts underlying the analogy are themselves meaningful to the person. Likewise, a computer model of analogy should offer some account of how the underlying symbols and structures that represent concepts in an analogy acquire meaning themselves, in addition to an account of the structure-mapping processes involved. That is, the structures that the program uses to represent analogies should be meaningful *to the program itself*. This is essentially the familiar symbol-grounding problem (Harnad, 1990), recast in analogical guise.

Some connectionist models of analogy have attempted to address this problem by moving away from the use of symbolic representations of source and target situations. Much recent work has focused on the use of distributed encoding techniques such as Plate's holographic reduced representations (Plate, 1994, 1998), Kanerva's binary spatter code (Kanerva, 1996, 1998), or Smolensky's tensor products (Smolensky, 1990). Examples of such models include *Drama* (Eliasmith & Thagard, 2001) and the STAR models of Halford et al. (1994) and Wilson et al. (2001). All of these approaches encode explicitly-structured representations of source and target situations as distributed activation patterns, which are suitable for processing by connectionist networks. These representations can be manipulated in a holistic fashion, without having to be decomposed into their constituent components (Chalmers, 1990; Chrisman, 1991; Blank, Meeden, & Marshall, 1992). However, currently the representations used by these models do not acquire their meaning internally through the system's own perceptions or through learning. Instead, meaning is imposed from outside the system through an essentially arbitrary assignment of semantics to the patterns of activation that serve as the constituent building blocks of representations. The hope is that eventually these systems will be able to use learned patterns based directly on sensory stimuli—instead of arbitrary patterns—as representational building blocks, which will make the representations meaningful to the system itself.

Blank's (1997) Analogator model attempts to integrate learning and analogy-making into a single connectionist framework using distributed representations based on tensor products. Analogator learns to make analogies between very simple visual scenes composed of geometric shapes, on the basis of spatial relationships such as *above* or *below*. Unlike the models mentioned previously, however, Analogator does not start with explicitly-structured representations. Instead, the system itself learns the meaning of spatial relationships by creating its own internal representations of analogies, through direct experience with visual scenes. In other words, the meaning of the underlying components of Analogator's analogies is acquired through the system's own perceptions. See Gasser (1993) for a more general discussion of perceptual grounding within the context of simple visual scenes.

In both Analogator and Metacat, perception is tightly interwoven with analogy-making. Analogator, however, focuses more on the learning of analogical *behavior* than on the explicit modeling of *concepts*. In contrast, Metacat emphasizes concepts and the ways in which they interact with perception, but does not attempt to model learning. Another difference is that Metacat's representations have a more symbolic flavor than the purely distributed representations created by Analogator. Nevertheless, the representations created by both models are much more closely tied to perception than the traditional predicate-calculus-based representations used by many of the models cited earlier.

2 Analogy-Making in an Idealized World

How can something as elusive as the meaning of concepts be modeled in a computer program? The approach taken by Copycat and Metacat is to start small, by eschewing real-world complexity in favor of a *microworld*—a tiny, idealized world designed to strip away as many distracting, surface-level details as possible from analogy-making while still preserving the fundamental essence of the phenomenon (Hofstadter, 1984). This philosophy differs from that of most other current computer models of analogy, which typically operate on representations of "real-world" situations that are not grounded in the program's own perceptions. We believe, however, that this is a deep and important issue that should be tackled head-on, rather than being sidestepped or ignored. In our approach, we restrict the number of concepts available in the world, which makes it possible for our models to represent concepts in a very rich and dynamic way that ties them intimately to perception. A limited set of concepts, however, need not imply a limited set of interesting analogy problems. Despite the microworld's apparent simplicity, it harbors an exceedingly rich variety of subtle analogy problems, in which many surprisingly creative and non-obvious answers are possible.

The raw material of this world consists of 26 abstract objects, represented as lowercase letters for convenience, among which only three relations are meaningful: sameness, predecessorship, and successorship. All letters except ***a*** have an immediate predecessor, and all except ***z*** have an immediate successor. All other information pertaining to letters has been factored out, such as their shapes or semantic connotations. Analogies are stated in terms of short letter-strings (called the *initial string*, the *modified string*, and the *target string*, respectively), which can be thought of as idealized situations. For example: "If ***abc*** changes to ***abd***, how does ***mrrjjj*** change in an analogous way?" Or, more succinctly:

abc ⇒ ***abd***
mrrjjj ⇒ ***?***

Most people, on seeing this problem for the first time, answer ***mrrkkk*** or ***mrrjjk*** (Mitchell, 1993). The rightmost component of ***abc*** (the letter ***c***) is perceived as changing to its successor, so doing the "same thing" to ***mrrjjj*** amounts to changing the rightmost component of ***mrrjjj*** to its successor—either ***jjj*** viewed as a chunk, or just the rightmost letter ***j***. There are, however, many other possible answers to this problem, which people tend to give less often, including:

- ***mrrjjd*** (change the rightmost letter literally to ***d***)
- ***mrrddd*** (change the rightmost chunk to ***d***'s)

- ***mrrjjj*** (change just the ***c***’s, of which there are none)
- ***mrrjkk*** (view ***mrrjjj*** as ***mr–rj–jj*** and change the rightmost pair to its successor)
- ***mrrjdd*** (view as ***mr–rj–jj***, but change the rightmost pair to ***d***’s)
- ***mrsjjj*** (change the third letter to its successor)
- ***mrdjjj*** (change the third letter to ***d***)
- ***mrsjjk*** (view as ***mrr–jjj*** and change the third letter of each chunk to its successor)
- ***mrskkk*** (change all letters after the first two to their successors)
- ***mssjjj*** (change every occurrence of the third letter to its successor)
- ***mrrjjjj*** (view ***mrrjjj*** abstractly as *1–2–3* and increase the rightmost length by one)
- ***mrrkkkk*** (view as *1–2–3* but change both the length and letters of the rightmost chunk)
- ***abd*** (change the whole string literally to ***abd***)
- ***abbddd*** (change the letters to ***a***’s, ***b***’s, and ***d***’s but retain the *1–2–3* structure)
- ***mrk*** (change ***j***’s to ***k***’s but make everything single letters)
- ***mrd*** (change ***j***’s to ***d***’s but make everything single letters)

Clearly, some of these answers are more obvious or plausible than others, but each one is defensible, and makes more sense than a completely random response such as ***pxznntg***. There is, however, no single, indisputably “correct” answer. In fact, a wide range of answers is possible for almost every conceivable problem in this world. The subtlety and richness of analogy-making has not been sacrificed at the expense of simplicity; on the contrary, it has been brought into focus more clearly precisely because of the world’s austerity.

It is also important to stress the intended *universality* of the microworld. “Letters” here are really nothing more than instances of abstract, atomic categories, among which only a small set of relations are meaningful (*i.e.*, successorship, predecessorship, and sameness). It is therefore misleading to regard Copycat’s or Metacat’s analogies as being about alphabetical strings of letters *per se*. Rather, strings should be viewed as representing idealized situations involving abstract categories and relations. The architecture of Copycat is “configured” so that these categories and relations mirror our intuitive notions about successorship, predecessorship, and sameness among letters of the alphabet, but this need not be the case. A different configuration could in principle be used, reflecting a different set of abstract relationships, without significantly altering the basic model. In fact, a program similar to Copycat, called Tabletop, models spatial aspects of high-level perception within a different domain: that of ordinary objects on a table, such as cups, glasses, and silverware (French, 1995; Hofstadter & FARG, 1995). Important differences exist between Copycat and Tabletop, but the two programs can be regarded essentially as different instantiations of a single underlying architecture, each of which operates in an abstract world of categories and relations. Copycat is configured so that these categories and relations reflect properties of letters of the alphabet, while Tabletop is configured so that they reflect properties of objects on a table.

Copycat's microworld is sometimes criticized as being unable to represent analogies between different domains of knowledge. So-called "cross-domain" analogies—for example, between the solar system and the Rutherford-Bohr model of the atom, or between water flowing through a pipe and heat flowing through a metal bar (Gentner, 1983; Holyoak & Thagard, 1989; Falkenhainer et al., 1990)—typically involve source and target situations characterized by very different kinds of "real-world" concepts. According to this view, the true power of analogy comes from being able to map quite different domains onto one another, allowing a transfer of knowledge to occur between them. In contrast, it is argued, since Copycat's source and target situations are restricted to letter-string concepts only, the model is "domain-specific", and hence fails to capture the most important aspects of analogical processing. According to Forbus, Gentner, Markman, and Ferguson (1998):

> The most dramatic and visible role of analogy is as a mechanism for conceptual change, where it allows people to import a set of ideas worked out in one domain into another. Obviously, domain-specific models of analogy cannot capture this signature phenomenon. . . . If we are correct that the analogy mechanism is a domain-independent cognitive mechanism, then it is important to carry out research in multiple domains to ensure that the results are not hostage to the peculiarities of a particular micro-world.

However, such a hasty conclusion overlooks the principle of universality at the core of Copycat's microworld. We fully agree that analogy is a very general, domain-independent cognitive mechanism. Indeed, this is the fundamental reason why we have chosen an abstract microworld as our framework for modeling analogy. Since the "letters"—as far as the program is concerned—are really just atomic categories linked by abstract relationships, there is in principle no reason why idealized versions of "cross-domain" analogies cannot be constructed within this world as well.

For example, the answer ***mrrjjjj*** to the earlier problem could be interpreted as just such an analogy. On the surface, different sets of concepts apply to the situations represented by the strings ***abc*** and ***mrrjjj***. In an abstract sense, these strings can be viewed as situations taken from two very different domains, each of which encompasses a distinct subset of the concepts available in the larger "universe" of the letter-string microworld. The concept of *successor*, for instance, is relevant to ***abc*** but not (at first glance) to ***mrrjjj***, while the concept of *group* plays a central role in ***mrrjjj***. If the two situations are looked at in the right way, however, by seeing the string ***mrrjjj*** in terms of group-lengths rather than letter-categories, the idea of successorship can be transferred over from the first situation to the second, resulting in a kind of mini paradigm shift that reveals the parallel *1–2–3* successorship structure of ***mrrjjj***, which consequently leads to the answer ***mrrjjjj***. Of course, both of these "domains" involve concepts from Copycat's letter-string world, but the crucial point is that they involve *different subsets* of concepts, just as the domains of "cross-domain" analogies from the real world involve different subsets of concepts taken from the larger universe of real-world concepts and relationships.

In fact, on closer examination, the distinction between different domains is often far from clear. For instance, Holyoak and Thagard (1995) discuss a complex analogy between World War II and the 1991 Persian Gulf War. Should this analogy be regarded as a "cross-domain" analogy, or as an analogy between two situations within the common domain of military conflicts? What about the analogy between the solar system and the Rutherford-Bohr atom? Does this analogy involve two distinct domains (*i.e.*, the domain of atomic physics and the domain of astronomy), or the single domain of scientific theories? In our view, the purported distinction between "cross-domain" and "intra-domain" analogies, as well as the distinction between "domain-general" and "domain-specific" models of analogy, is artificial, and depends on the particular definition of the domains involved, which in turn depends on how we as researchers choose to carve the world up into categories. The power of a microworld derives precisely from its ability *in principle* to model any number of different subdomains of the real world within a common abstract framework.

xyz family

abc ⇒ abd ***xyz ⇒ xyd***	***abc ⇒ abd*** ***xyz ⇒ wyz***	***abc ⇒ abd*** ***xyz ⇒ dyz***
rst ⇒ rsu ***xyz ⇒ xyu***	***rst ⇒ rsu*** ***xyz ⇒ wyz***	***rst ⇒ rsu*** ***xyz ⇒ uyz***

mrrjjj family

abc ⇒ abd ***mrrjjj ⇒ mrrkkk***	***abc ⇒ abd*** ***mrrjjj ⇒ mrrjjjj***
xqc ⇒ xqd ***mrrjjj ⇒ mrrkkk***	***xqc ⇒ xqd*** ***mrrjjj ⇒ mrrjjjj***

eqe family

eqe ⇒ qeq ***abbba ⇒ baaab***	***eqe ⇒ qeq*** ***abbba ⇒ aaabaaa***
eqe ⇒ qeq ***abbbc ⇒ qeeeq***	***eqe ⇒ qeq*** ***abbbc ⇒ aaabccc***

Figure 1: Three families of letter-string analogies

3 Three Families of Analogy Problems

Fig. 1 shows three families of analogy problems, which will be used as examples throughout the remainder of the paper to illustrate the principal mechanisms and capabilities of Metacat. These problems give a sense of the types of parallels and distinctions that can be made between analogies in the letter-string world. Each family consists of two distinct (but similar) analogy problems, with horizontal rows showing a set of possible answers for each problem.

The first family consists of the problem ***abc ⇒ abd; xyz ⇒ ?*** and its variant ***rst ⇒ rsu; xyz ⇒ ?*** (top of Fig. 1). Viewing ***c*** as changing to its successor in ***abc ⇒ abd; xyz ⇒ ?*** suggests changing ***z*** to its successor. However, this is not possible in the letter-string world, so one is forced to try something else. One way out is to adopt a literal-minded approach and change ***z*** to ***d***, yielding ***xyd***. On the other hand, if the alphabetic symmetry between ***a*** and ***z*** is noticed, then the more abstract answer ***wyz*** may come to mind, based on seeing ***abc*** and ***xyz*** as mirror images of each other wedged against opposite ends of the alphabet. In this symmetric interpretation of the problem, doing the “same thing” to ***xyz*** means changing the *leftmost* letter to its *predecessor* instead of changing the rightmost letter to its successor. Many people consider this answer to be more elegant and compelling than ***xyd***.

Now consider the variant problem ***rst*** ⇒ ***rsu; xyz*** ⇒ ***?***. The literal-minded answer ***xyu*** and the symmetric answer ***wyz*** are both possible, and arise for the same reasons as in the previous problem—with one important difference. In this problem there is far less justification for seeing ***rst*** and ***xyz*** as mirror images of each other, unlike in the case of ***abc*** and ***xyz***, with their strong ***a***-***z*** symmetry, which makes the answer ***wyz*** a weaker analogy here than in the previous problem. While it could be argued that ***wyz*** is still a better analogy than ***xyu*** in this problem, it is clearly not *as* superior to ***xyu*** as ***wyz*** was to ***xyd*** in the previous problem. The two ***wyz*** analogies, therefore, are quite different in character, even though they involve identical answers. Indeed, the presence or absence of alphabetic symmetry is the fundamental difference between them. The literal-minded answers ***xyd*** and ***xyu***, on the other hand, represent essentially identical analogies, despite their surface-level differences.

Two other answers are also worth mentioning. The answer ***dyz***, although perhaps a bit far-fetched, is certainly possible for ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***. This answer depends on noticing the abstract symmetry between ***abc*** and ***xyz*** (and thus changing the ***x*** in ***xyz*** instead of the ***z***) but taking a very literal-minded view of ***abc*** ⇒ ***abd*** (thus changing ***x*** to ***d*** instead of to its predecessor). The answer ***uyz*** to the problem ***rst*** ⇒ ***rsu; xyz*** ⇒ ***?*** arises in a similar fashion, except that once again, there is no good reason to see ***rst*** and ***xyz*** as mirror images in the first place. This blend of abstractness and literal-mindedness makes both of these answers seem incoherent. It could even be argued that since ***abc*** and ***xyz*** are completely symmetric in every way, while ***rst*** and ***xyz*** are not, changing ***x*** to ***d*** in ***abc*** ⇒ ***abd; xyz*** ⇒ ***?*** is even *more incoherent* than changing ***x*** to ***u*** in ***rst*** ⇒ ***rsu; xyz*** ⇒ ***?***, making ***dyz*** a more incoherent analogy than ***uyz***. Just like the two ***wyz*** analogies, the key distinction between ***dyz*** and ***uyz*** is the presence or absence of alphabetic symmetry. In other words, the *way* in which the two ***wyz*** analogies are different is analogous to the way in which the ***dyz*** and ***uyz*** analogies are different. Here we have a simple example of a “meta-analogy” in the letter-string microworld.

The second family of analogies consists of the answers ***mrrkkk*** and ***mrrjjjj*** to the pair of problems ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?*** and ***xqc*** ⇒ ***xqd; mrrjjj*** ⇒ ***?*** (middle of Fig. 1). Each of these analogies relies on seeing the target string ***mrrjjj*** in terms of its three components ***m***, ***rr***, and ***jjj***—corresponding to the three letters of the initial string—and on viewing the rightmost letter of the initial string as changing to its successor. The rightmost component of ***mrrjjj*** (the ***jjj*** group) accordingly changes to its successor, yielding ***mrrkkk*** if ***mrrjjj*** is viewed in terms of letter-categories (as *m–r–j*), or ***mrrjjjj*** if it is viewed in terms of group-lengths (as *1–2–3*).

In the problem ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***, the answer ***mrrjjjj*** represents a stronger analogy than ***mrrkkk***, because viewing ***mrrjjj*** as *1–2–3* reveals an abstract similarity between the target string's structure and the parallel *a–b–c* structure of the initial string. On the other hand, the answer ***mrrkkk*** makes for the stronger analogy in the problem ***xqc*** ⇒ ***xqd; mrrjjj*** ⇒ ***?***. Unlike ***abc***, the string ***xqc*** possesses no internal successorship structure, so viewing ***mrrjjj*** in an unstructured way as *m–r–j* more closely parallels ***xqc***, while viewing it as *1–2–3* amounts to being unnecessarily "clever". In short, the two ***mrrkkk*** answers are actually quite different in character, as are the two ***mrrjjjj*** answers.

The third family of analogies consists of the problem ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***?*** and its variant ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?*** (bottom of Fig. 1). In these problems, ***eqe*** can be viewed as "turning itself inside-out" by swapping the letter-categories of its constituent letters to yield ***qeq***. If ***abbba*** is viewed as ***a–bbb–a***, corresponding to the three letters of ***eqe***, then a natural way of doing the same thing to ***abbba*** is simply to swap the letter-categories of the components, yielding ***baaab***. This approach, however, leads to a "snag" in the case of ***abbbc***, because swapping *three* letter-categories makes no sense. One way around this difficulty is to view the letters of ***eqe*** as changing individually to ***q***, ***e***, and ***q***, instead of getting collectively swapped. Changing ***abbbc*** in an analogous way would then amount to changing its three components to ***q***, ***eee***, and ***q***, yielding the answer ***qeeeq***.

A more elegant way of avoiding the snag is to perceive ***abbbc*** abstractly as *1–3–1* and then swap the *lengths* of the components instead of the letter-categories, yielding ***aaabccc***. This is reminiscent of the answer ***mrrjjjj*** to the problem ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***.

On the other hand, we can do this in the problem ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***?*** as well, swapping lengths instead of letter-categories to yield ***aaabaaa***. However, as in the earlier analogy ***xqc*** ⇒ ***xqd; mrrjjj*** ⇒ ***mrrjjjj***, viewing ***abbba*** as *1–3–1* is unnecessarily "clever", since swapping letter-categories works just fine. Thus the difference between the answers ***baaab*** and ***aaabaaa*** in the problem ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***?*** is like the difference between the answers ***mrrkkk*** and ***mrrjjjj*** in the problem ***xqc*** ⇒ ***xqd; mrrjjj*** ⇒ ***?***, because in both problems viewing the target string abstractly actually makes for a *weaker* analogy.

In contrast, viewing the target string abstractly in the problems ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?*** and ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?*** makes for a *stronger* analogy in each case—although not for precisely the same reasons. In the case of ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?***, viewing ***abbbc*** as *1–3–1* has the added benefit of enabling a snag to be avoided, whereas no snag arises in ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***. In other words, the answer ***aaabccc*** is strong for both pragmatic and aesthetic reasons, while ***mrrjjjj*** is strong for aesthetic reasons only. Likewise, ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***aaabaaa*** is a weaker analogy than ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***aaabccc***, even though both involve seeing the target string as *1–3–1*, because paying attention to lengths is justified in the latter analogy on account of the snag, but not in the former.

4 The Copycat Model

This section summarizes the architecture and processing mechanisms of Copycat, which serve as the foundation for Metacat's architecture. Several important limitations of the original model, which have been addressed in Metacat, are also pointed out.

The Copycat architecture has been discussed at length elsewhere (Mitchell, 1993; Hofstadter & FARG, 1995), so details will be omitted here. Briefly, the program consists of a long-term memory for concepts, called the *Slipnet*, together with a short-term memory for perceptual structures, called the *Workspace*. The Slipnet is a semantic network of nodes representing concepts about the letter-string world (see Fig. 2), with weighted links between nodes encoding the strength of associations between concepts. Some links are *labeled* by particular nodes, and may stretch or shrink according to the activation of the label node, allowing the Slipnet to dynamically adapt to the perceptual context at hand. Some concept nodes, shown capitalized in Fig. 2, represent categories of other concepts. For example, *left* and *right* are both instances of the more abstract *Direction* category, and the concepts *letter* and *group* are both instances of *ObjectCategory*.

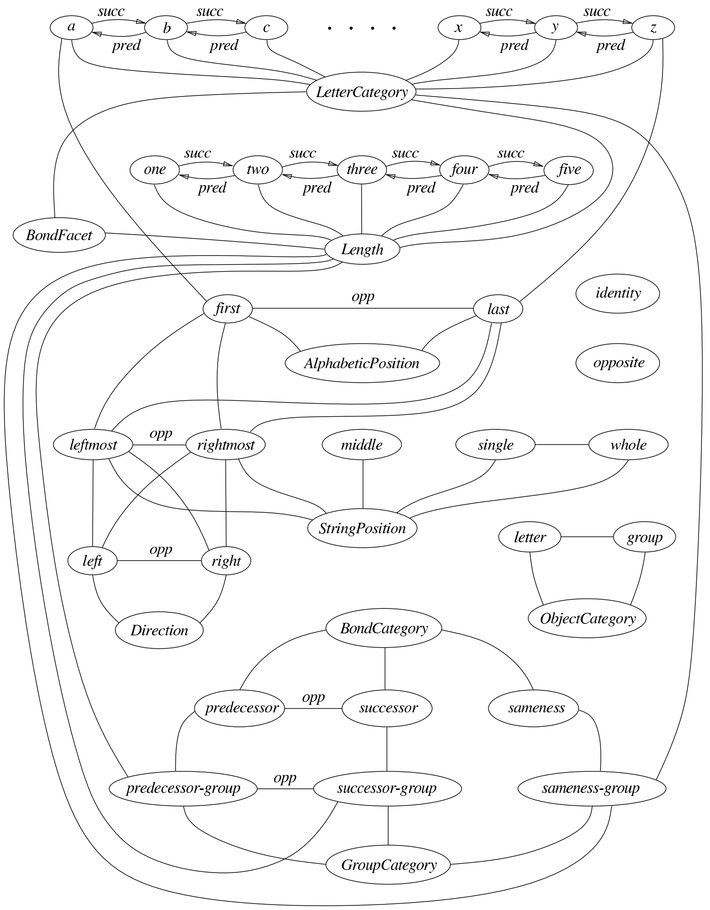


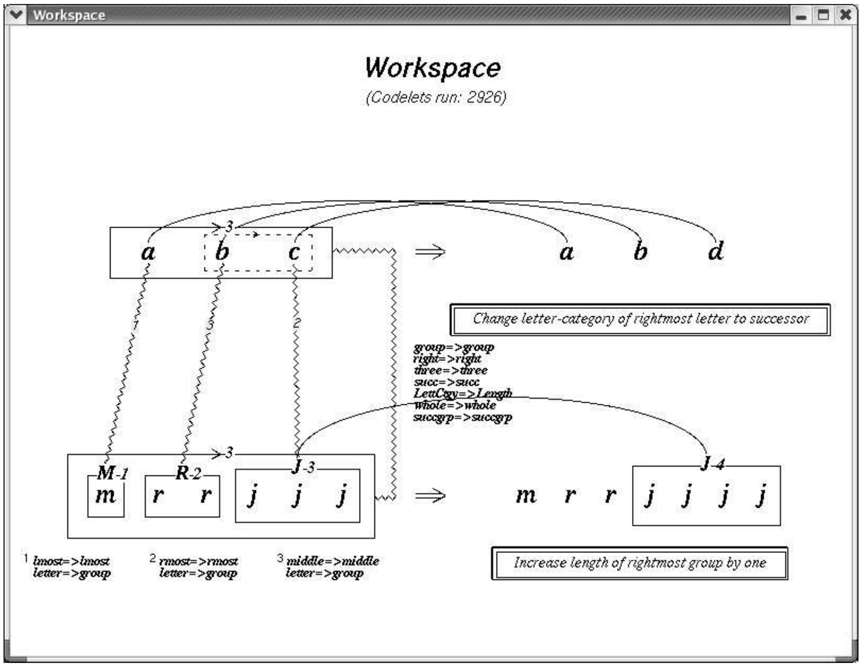
Figure 2: The Slipnet

The Workspace is the site of subcognitive processing activity. In the Workspace, small nondeterministic computational agents called *codelets* examine the letters of an analogy problem and attempt to build a coherent set of structures around the letters, representing a particular interpretation of the problem. Codelets look for sameness, successor, or predecessor relationships among letters, possibly chunking them together into *groups* based on a common relationship (for example, creating a "sameness group" from the three ***j***'s in ***mrrjjj***, or chunking the individual letters of ***abc*** into a single "successor group"). The program's high-level behavior emerges in a bottom-up fashion from the collective actions of many codelets working in parallel, analogous to the way in which an ant colony's high-level behavior emerges from the individual behaviors of the underlying ants, with no centralized locus of control.

In general, the letter-strings of an analogy problem can be viewed in many different ways, giving rise to a vast space of potential configurations of Workspace structures. In order to discover a good configuration in a reasonable amount of time, many potential pathways through "interpretation space" must be explored simultaneously, with proportionally more attention being paid to promising pathways than to those that don't seem to be leading anywhere interesting. This type of differential parallelism, called the *parallel terraced scan*, is one of the central ideas of the Copycat architecture.

To achieve this differential effect, structures are built in stages by chains of codelets. At first, a codelet simply proposes a new structure as a possibility. The proposed structure is then evaluated by other codelets at later stages in the chain. If the structure seems promising enough, it gets built, and acquires a *strength* value indicating how well it fits into its surrounding context. By distributing structure creation over several interleaved stages, many different pathways can be explored in parallel. In addition, every codelet has an *urgency* value that reflects the estimated promise of the pathway it is exploring. Codelets are selected to run, probabilistically, on the basis of their urgencies. Therefore promising regions of the search space tend to be explored more quickly and to a greater depth, on average, than less promising regions.

Fig. 3 shows a set of perceptual structures at the end of a run on the problem ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***. Several groups can be seen, including one built from other groups and one consisting of the single letter ***m***. One proposed group (shown as a dashed structure), which was being tentatively explored but had not yet been built by codelets, can also be seen. In this run, the program has perceived the abstract *1–2–3* successorship of ***mrrjjj*** and mapped this onto the *a–b–c* successorship of ***abc***. Horizontal and vertical structures called *bridges* show the correspondences between analogous components of each situation. For example, the ***c–jjj*** bridge indicates that, in this interpretation of the problem, ***c*** and ***jjj*** play analogous roles in their respective strings. *Concept-mappings* associated with each vertical bridge can also be seen (they are not shown for horizontal bridges). For example, *rightmost* ⇒ *rightmost* and *letter* ⇒ *group* are associated with the ***c–jjj*** bridge, since ***c*** and ***jjj*** are both rightmost objects, but one is a letter and the other is a group. Non-identity concept-mappings such as *letter* ⇒ *group* are called *slippages*, and serve as the basis for generating an answer. For instance, the *LetterCategory* ⇒ *Length* slippage underlying the high-level bridge between ***abc*** and



Change letter-category of rightmost letter to successor

Increase length of rightmost group by one

Figure 3: An interpretation of the problem ***abc* ⇒ *abd; mrrjjj* ⇒ ?**

mrrjjj reflects the fact that the "successorship fabric" of ***abc*** is based on letter-categories, while that of ***mrrjjj*** is based on group-lengths. This slippage, together with the *letter ⇒ group* slippages, leads the program to produce the answer ***mrrjjjj*** by changing the length of the rightmost group in ***mrrjjj*** to its successor, instead of changing the letter-category of the rightmost letter as was done in ***abc***.

Concepts in the Slipnet influence the search for a mutually-consistent set of perceptual structures by acquiring activation in response to codelet activity in the Workspace. This activation, which may spread to neighboring concepts, strongly affects the nondeterministic decisions made by codelets, resulting in top-down pressure that guides the program in its search for a good interpretation of a problem. Each concept has a fixed *conceptual depth* value associated with it, which represents its intrinsic degree of abstractness or generality. The activation of a concept gradually decays at a rate that depends on its conceptual depth, with highly abstract concepts such as *opposite* tending to decay more slowly than shallow, surface-level concepts such as *d*.

To be sure, Slipnet concepts come nowhere close to capturing the full power and fluidity of human concepts. Nevertheless, there is a sense in which they *are* genuinely meaningful entities—not simply passive, static tokens manipulated by the program. For example, a Slipnet node such as *successor* responds to situations in a continuous, context-dependent way, with its level of activation changing to reflect the current degree of perceived relevance of the idea of successorship in the problem at hand. A wide range of superficially dissimilar situations, represented abstractly as letter-strings, can in principle activate it—strings such as ***abc***, ***ijk***, ***pqrst***, ***iijjkk***, ***mrrjjj***, ***xxsssbbbb***, ***axypqr***, and ***aababcabcd***. Under the right circumstances, all of these strings can be interpreted by the program as examples of successor groups. The semantics of the *successor* node arises precisely from the way in which this node responds to different situations perceived in the "environment" of the letter-string microworld. In other words, its meaning is determined by its behavior within the system, not by a particular interpretation imposed on it from outside the system. Given Copycat's ability to flexibly recognize a wide range of instances—some fairly abstract—of the same concept, it seems reasonable to say that the program's concepts have at least some small degree of meaningfulness, or genuine semantics, within the confines of its tiny, idealized world. See Hofstadter and FARG (1995, Chapter 6) for an in-depth discussion of this point.

Slipnet concepts also serve as the basic building blocks for other structures called *rules*, which describe how strings change.[1] For example, in Fig. 3, two rules can be seen. The top rule, *Change letter-category of rightmost letter to successor*, describes how the program views ***abc*** as changing to ***abd***. The bottom rule, *Increase length of rightmost group by one*, describes ***mrrjjj*** ⇒ ***mrrjjjj***. Internally, rules are structured collections of Slipnet nodes. Outwardly, they are displayed as short English phrases for readability, but this is really just a surface-level "veneer" masking the underlying conceptual representation. For instance, the top rule in Fig. 3 is composed of the concepts *LetterCategory*, *StringPosition*, *rightmost*, *letter*, and *successor*. The bottom rule is composed of *Length*, *StringPosition*, *rightmost*, *group*, and *successor*.

Copycat places severe restrictions on the types of changes that are allowed in the initial string. At most, one letter is allowed to change, such as in ***abc*** ⇒ ***abd***. For instance, all of the analogies in the ***eqe*** family shown in Fig. 1 are beyond Copycat's ability to handle. This is because the development of Copycat concentrated on designing mechanisms for perceiving *similarities* between the initial string and the target string via bridges and slippages, rather than on characterizing *dif-*

[1] This usage of the term "rule" differs significantly from the traditional AI meaning of the term. Rules in Copycat and Metacat are completely unrelated to the "if-then" rules used in expert systems or other rule-based production systems.

ferences between strings via rules. Developing robust mechanisms for mapping the initial string to the modified string, and for creating rules based on this mapping, was postponed to a later phase of the project. These mechanisms have now been extended and generalized in Metacat to handle arbitrary mappings between strings, enabling a much wider class of string changes to be described by rules, including ***eqe*** ⇒ ***qeq***. See Marshall (1999, Chapter 3) for a full discussion of Metacat's new rule-building mechanisms.

The overall degree of Workspace organization is measured by a number from 0 to 100 called the *temperature*. This number is a function of the total quality of structures present in the Workspace—where the quality of a structure is determined by its strength. Temperature also regulates the amount of randomness used by codelets in making decisions. In other words, temperature plays both a passive and an active role. At high temperatures, when few Workspace structures exist, decisions are made in a highly random manner, since not much is yet known about the problem. However, as relationships among the letters are noticed and new structures are built, the temperature falls, and Copycat begins to gain "confidence" in the emerging interpretation of the problem. At lower temperatures, decisions are made less randomly, being more strongly biased by the estimated quality of newly emerging structures, all of which compete for the attention of codelets. At very low temperatures, codelets pay attention to only the most promising structures, and decisions become largely deterministic. Thus the type of strategy used by the program to explore its search space gradually shifts from being very diffuse and stochastic at high temperatures to being very focused and deterministic at low temperatures.

To reiterate, processing in Copycat is driven by a large number of fine-grained probabilistic decisions that depend on the current temperature. These decisions may cause new structures to be built or existing structures to be destroyed, which in turn changes the temperature and consequently affects processing, forming a feedback loop. Temperature thus serves as a very simple form of self-watching in Copycat, since it enables the program to regulate its own behavior to a limited extent. In other words, tying the stochastic activity of codelets to the temperature makes the program sensitive to its own actions.

This type of self-watching, however, is very primitive and unfocused. Temperature allows Copycat to respond to its immediate situation in a reactive way, but the program remains oblivious to longer-term patterns that arise in its processing over time. This can result in very unhumanlike behavior. For instance, when presented with the problem ***abc* ⇒ *abd; xyz* ⇒ *?***, Copycat usually attempts to change *z* to its successor, which is impossible in the program's microworld. It hits a snag, and is forced to try something else. However, it typically ends up just trying the same thing over and over again, often as many as ten or twenty times in a row before stumbling by chance on an alternative answer (such as ***xyd***). Unlike people, the program is unable to recognize when it has fallen into a repetitive and futile pattern of behavior. Because it has no memory of its past experiences, it cannot recognize that it has already encountered some situation before, or tried the same set of actions in response.

5 From Copycat to Metacat

Since Copycat is incapable of remembering its past actions or experiences, it has no knowledge of how it arrives at its answers, and is therefore unable to explain the rationale behind the analogies it makes, or why one analogy is better or worse than another. In contrast, Metacat's architecture includes several new components and mechanisms that allow the program to monitor itself, enabling it to recognize, remember, and recall patterns that occur in its "train of thought" as it makes analogies. To do this, Metacat creates an explicit temporal record of the most important processing events that occur during a run. This record is continually examined by codelets for patterns, in much the same way that codelets examine letter-strings for patterns. It also provides the basis for constructing an abstract description of an answer in terms of the key concepts and events that led to its discovery. Consequently, Metacat is able to construct much richer representations of analogies, enabling it to compare and contrast them in an insightful way. Furthermore, by monitoring its own processing, Metacat can recognize when it has become stuck in a rut, enabling the program to break out of the rut by explicitly focusing on ideas other than the ones that seem to be leading nowhere. This capability affords the program a powerful degree of self-control.

The remainder of the paper describes the architecture of Metacat, focusing on the ways in which it extends the capabilities of Copycat, and analyzes several sample runs that illustrate different aspects of the model. Since Metacat is an extension of Copycat, its architecture includes the Workspace, the Slipnet, and the mechanisms for codelet processing. It also includes three new components: the *Episodic Memory*, the *Themespace*, and the *Temporal Trace*.

5.1 The Episodic Memory

Metacat stores descriptions of analogies in its long-term Episodic Memory. When a new answer is found, an *answer description* is created from the information available in the temporal record and the Workspace. This description includes the four letter-strings of the analogy, as well as the rules, bridges, slippages, and other structures that give rise to the answer. Other structures called *themes* are also included, which describe the key underlying concepts of the analogy.

Themes provide a basis for comparing and contrasting answers, as well as a metric for judging the degree of similarity between them. For instance, when Metacat makes a new analogy, it may be reminded of a similar analogy it has seen in the past if the themes associated with the newly-created answer description, acting as a memory retrieval cue, match those of some previously stored answer description sufficiently well. In effect, the pattern of themes in an answer description serves as an index for storing and retrieving an answer from memory.

In addition to remembering answers, Metacat also remembers the snags that it encounters while solving problems. On hitting a snag for the first time, the program creates a new *snag description* that characterizes the failure in terms of the themes and other structures involved, which it then stores in the Episodic Memory. Snag descriptions can be compared on the basis of their themes, enabling Metacat to evaluate the similarity of different failure situations. Furthermore, comparing the themes of snag descriptions and answer descriptions can provide clues as to how failures can be avoided in certain problems.

5.2 Themes and the Themespace

Themes are short-term memory structures that describe the characteristics of mappings between letter-strings in a high-level, abstract way. They are composed of Slipnet concepts, and are created in Metacat's Themespace in response to structure-building activity in the Workspace. For example, in the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, if a crosswise mapping is built between ***abc*** and ***xyz*** as a result of noticing the alphabetic symmetry between ***a*** and *z*, a theme composed of the concepts *AlphabeticPosition* and *opposite* will be created. A *StringPosition: opposite* theme will also be created, representing the idea that objects in opposite positions in their respective strings correspond to one another, as expressed by the bridges ***a–z*** and ***c–x***. On the other hand, if the ***a-z*** symmetry is not noticed and a parallel mapping consisting of the bridges ***a–x***, ***b–y***, and ***c–z*** is built instead, no *AlphabeticPosition* theme would be created. In this case, a *StringPosition: identity* theme would describe the parallel mapping. Thus themes capture the essential aspects of an analogy by concisely summarizing how the letter-strings are perceived in relation to one another.

Like Slipnet concepts, themes take on varying levels of activation, reflecting the extent to which the ideas they represent play a role in the program's current perception of the problem. In this sense they behave as passive representational structures. However, in certain situations, to be explained below, themes can exert strong top-down *thematic pressure* on perceptual processing. This pressure, which can be turned on or off by the program itself, selectively weakens or strengthens existing structures in the Workspace, and may cause codelets to focus on building specific types of new structures. In fact, unlike Slipnet concepts, themes can assume both positive and negative levels of activation. With thematic pressure turned on, positively-activated themes encourage the creation of structures compatible with the ideas represented by the themes. Negatively-activated themes, on the other hand, *discourage* the creation of such structures; instead, they promote the creation of alternative structures incompatible with themselves. Thus themes act like a set of "knobs" that can be used to focus the attention of the program on specific sets of ideas. By twisting the knobs—that is, by varying the pattern of theme activations under thematic pressure—Metacat's perceptual processing can be steered in particular directions, guided by the ideas explicitly represented by the themes.

5.3 The Temporal Trace

The *Temporal Trace* (or the *Trace* for short) serves as the locus for self-watching in Metacat. Like the Workspace and Themespace, it is a short-term memory that stores information over the course of a single run. The Trace stores an explicit temporal record of the most important *processing events* that occur during problem solving. Examples of such events include the strong activation of a theme or concept, making a conceptual slippage, creating a new rule, hitting a snag, or discovering a new answer. Of course, a large number of events of all sizes occur during the processing of almost any analogy problem, ranging from local "micro" events such as individual codelet actions to global "macro" events such as the discovery of new answers. However, only those events above a threshold level of importance get represented in the Trace. This allows Metacat to filter out all but the most significant events, giving the program a very selective, high-level view of what it is doing.

One way to appreciate the abstract, chunked nature of the information in the Trace is to consider the typical number of steps that occur during a run of Metacat. This depends on the level of granularity used to describe steps. At a very fine-grained level of description, where each step corresponds to an action performed by a single codelet, a run may consist of many hundreds or even thousands of steps. At this level of description, no two runs are ever *exactly* the same, even if they involve the same letter-strings (unless both runs start with the same random number seed). On the other hand, at the level of description of the Trace, a typical run consists of a few dozen steps. At this level of granularity, each step corresponds to a single event recorded in the Trace, and represents the actions of many codelets.

For example, Fig. 4 shows the contents of the Trace after a run on the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, in which the program, after trying unsuccessfully a couple of times to change *z* to its successor, answered ***xyd***. The events that occurred during the run appear in chronological order from left to right. This run involved a total of 1,558 codelets, but the high-level picture shown in the Trace consists of just twelve events, which represent the "major milestones" encountered along the way in the program's search for an answer. For instance, the Slipnet concept *identity* got activated early on, due to the program perceiving the ***a***'s and ***b***'s in ***abc*** and ***abd*** as corresponding to one another. This was followed by the chunking of ***abc*** and ***xyz*** into predecessor groups going in the same direction (both to the left). Next, the rule *Change letter-category of rightmost letter to successor* was created for describing ***abc*** ⇒ ***abd***, which led inevitably to a snag. In the aftermath of the snag, another rule was created (*Change letter-category of rightmost letter to 'd'*), and ***abc*** and

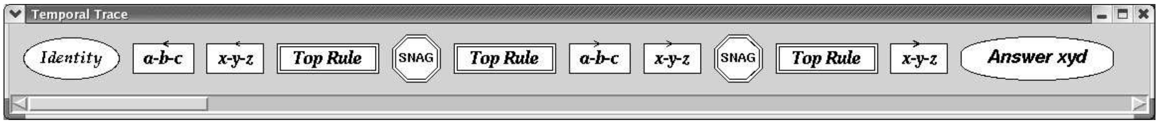


Figure 4: The temporal record of a run on the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***

xyz were reperceived as *successor* groups (again going in the same direction—only this time to the right). However, the program then attempted to use the first rule again, resulting in another snag. Finally, after creating a third rule and again perceiving ***xyz*** as a successor group, the program found the answer ***xyd***.

Once processing events have been explicitly represented in the Temporal Trace, they are themselves subject to examination by codelets. This allows Metacat to perceive patterns in its own behavior in much the same way that Copycat perceives patterns in letter-strings: via codelets looking for relationships among perceptual structures. In Metacat's case, these perceptual structures include the "reified" processing events in the Trace. When a new answer is found, an answer description is created by examining the temporal record to see which events contributed to the answer's discovery.

This approach is similar in flavor to work on derivational analogy, in which the trace of a problem-solving session is stored in memory for future reference, together with a series of annotations describing the conditions under which each step in the solution was taken (Carbonell, 1986; Veloso & Carbonell, 1993; Veloso, 1994). In Metacat's case, however, the information in the Trace is used as the basis for constructing an abstract description of the answer found, rather than being permanently stored itself.

6 Pattern-Clamping and Self-Control

The Trace allows Metacat to monitor the processing activity in the Workspace at a very abstract and highly chunked level of description, enabling the program to "see" what it is doing during a run. Equally important, however, is the program's ability to *respond* to what it sees by clamping particular themes and concepts at high activation, resulting in strong top-down pressure on processing. Various types of *patterns*, consisting of sets of themes, concepts, or codelet urgencies, can be clamped by the program in response to events in the Trace. Clamping a pattern alters the probabilities that certain types of codelets will run, or that certain types of Workspace structures will be built, which may lead the program to revise its interpretation of a problem by reorganizing structures in the Workspace in accordance with the ideas represented by the pattern. Thus patterns serve as a "medium" through which the program is able to wield control over its own behavior.

When an event is recorded in the Trace, the themes most active at the time of the event are stored along with it. These themes serve as the event's *thematic characterization*. In the case of a snag event, the thematic characterization represents a failed way of interpreting the problem. For example, in solving ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, Metacat usually first perceives ***abc*** and ***xyz*** as going in the same direction, which leads to a snag whose thematic characterization includes the theme *StringPosition: identity*. If Metacat continues to hit the same snag over and over, a series of snag events will accumulate in the Trace, all with very similar thematic characterizations. This similarity may be noticed by codelets (the probability becoming higher as more snags accumulate), causing them to take action by clamping the "offending" themes, including *StringPosition: identity*, with strong negative activation. This encourages the program to explore alternative ways of interpreting the problem, which may subsequently lead it to discover other answers such as ***wyz***. In this way, Metacat can recognize its own repeated failures and respond accordingly.

Codelets called *Progress-watchers* are responsible for deciding whether or not to unclamp a clamped pattern. In general, the purpose of clamping a pattern is to catalyze a series of events that reorganize the perceptual configuration of the Workspace in some way. It is therefore better to wait until the structure-building activity occurring in the wake of a clamp has settled down before concluding that the clamp has "run its course". If a *Progress-watcher* codelet runs while a pattern is clamped, it examines the most recent event in the Trace to determine how much time has elapsed since the event occurred. If the amount of elapsed time is less than a minimum settling period, then the codelet simply fizzles, leaving the clamped pattern still in effect. On the other hand, if enough time has passed without any new important events having transpired, the codelet unclamps the pattern and then evaluates the amount of progress that was made since the clamp occurred. Depending on the amount of progress achieved, the codelet may decide to spawn a follow-up codelet to see whether a new answer can be made based on the newly-created structures.

The criteria for evaluating the success of a clamp can vary. Sometimes, the purpose of clamping a pattern is to promote the creation of specific types of Workspace structures. Other times, the purpose is to encourage the creation of structures of any type, so long as they are compatible with the clamped pattern. The progress achieved by a clamp can be measured by observing the number of structures that get built in the immediate aftermath of the clamp, and the extent to which they are compatible with the pattern.

If no patterns are clamped when a *Progress-watcher* codelet runs, then instead of checking on the progression of events in the Trace, the codelet checks on the current rate of structure-building activity in the Workspace. This activity is measured by a number from 0 to 100, which serves as a quick estimate of the "freshness" of the current structures in the Workspace. More precisely, it is an inverse function of the average age of the most recently created structures. Thus the activity level tends to remain high as long as new structures are being built, but eventually drops to zero in the absence of new structures.

If the activity level is zero, indicating that nothing much is happening in the Workspace, then Metacat may have arrived at an impasse in its search for answers to the current problem. This is not quite as bad as hitting a snag, but it still ought to prod the program into trying something different. However, in the case of an impasse, there is usually no clear set of "offending" structures or themes to pin the blame on, unlike in the case of a snag. Indeed, the impasse may well arise from a *lack* of appropriate structures, rather than from the existence of the "wrong" structures. Therefore, in the absence of Workspace activity, *Progress-watcher* codelets check to see whether particular types of new structures may be needed. If so, they may clamp a pattern of codelet urgencies in response, in an attempt to catalyze structure creation.

For example, a *Progress-watcher* might examine the quality of the rules that have been built so far. If no good rules yet exist, the codelet might try to encourage the creation of better rules by clamping a pattern of codelet urgencies that strongly increases the probability that rule-seeking codelets will run, while inhibiting other types of codelets. Eventually, other *Progress-watchers* will turn off the clamp once enough time has passed with no more events having been added to the Trace.

Since this particular clamp is only concerned with the creation of new rules, the amount of progress achieved is judged solely on the basis of the quality of the rules that get created in the clamp's wake (if any).

6.1 Answer Justification

Metacat's pattern-clamping mechanisms give it another important capability, which Copycat lacks. Unlike Copycat, Metacat is able to evaluate analogies suggested to it by the user, in addition to making analogies on its own. When provided with a specific answer to a problem, the program "works backwards" from the answer toward an understanding of why it makes sense. Once the answer has been understood, it can be compared and contrasted with other answers that the program has either been shown or has discovered on its own.

This type of "hindsight understanding" presents little difficulty for humans. People who are asked to solve the problem ***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***, for example, may not think of the answer ***mrrjjjj***, even when given an unlimited amount of time. However, as soon as this answer is suggested to them, they have no trouble seeing why it makes sense, even though they didn't think of it themselves. In a similar vein, suggesting the somewhat tongue-in-cheek answer ***abd*** usually elicits a few laughs, along with nodding agreement that it makes sense in a silly way, although few people give this answer on their own (Mitchell, 1993). This is not to say that *every* suggested answer can be readily understood in retrospect (for example, a person might never figure out the justification for an answer such as ***mssjjj***), but for many non-obvious answers, no additional explanation beyond just the answer itself is needed.

When Metacat runs in *justify mode*, it attempts to discover a way of interpreting the problem such that the given answer makes sense. To do so, it begins by building up perceptual structures among the letter-strings, as usual. This bottom-up approach, however, may lead it to build an inconsistent interpretation of the problem that does not support the answer in question. Nevertheless, examining parts of this interpretation may suggest new ideas to focus on. More precisely, an *Answer-justifier* codelet may compare the rule structures involved and, based on their differences, clamp a pattern of themes designed to reorganize the mapping between the initial string and the target string in a way consistent with the rules and the answer.

For example, when Metacat is asked to justify the answer ***wyz*** to the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, it usually starts out by building same-direction mappings between all of the strings. (Snags do not arise in justify mode, since the answer already exists.) In addition, the "top" rule *Change letter-category of rightmost letter to successor* describing ***abc*** ⇒ ***abd***, and the "bottom" rule *Change letter-category of leftmost letter to predecessor* describing ***xyz*** ⇒ ***wyz*** may be created. This state of affairs is shown in Fig. 5. Although the three string mappings are locally consistent when considered in isolation, together they do not make sense at a global level. The letters ***c*** and ***x*** are not seen as corresponding to each other, yet they are both identified by the rules as being the objects that change in their respective strings (the ***c*** to its successor and the ***x*** to its predecessor).

Comparing the two rules to each other, however, suggests the idea of *rightmost-leftmost* symmetry, as well as *successor-predecessor* symmetry. This idea can be captured by a pattern of themes including *StringPosition: opposite*, *Direction: opposite*, and *GroupType: opposite*.[2] Metacat can explore the ramifications of this idea by positively clamping these themes in the Themespace. The state of the Temporal Trace at the time of the clamp is shown above the Workspace in Fig. 5. As can be seen, clamping the pattern causes the concept of *opposite* in the Slipnet to become highly activated. The ensuing top-down thematic pressure strongly promotes the creation of new structures that support mapping ***abc*** and ***xyz*** onto each other in a crosswise fashion, and significantly weakens existing incompatible structures such as the ***a–x*** and ***c–z*** bridges. As a result, the original mapping between ***abc*** and ***xyz*** shown in Fig. 5 is swiftly reorganized by codelets into a new mapping consistent with the clamped themes.

Fig. 6 shows the final, globally consistent interpretation, in which ***c*** and ***x*** are seen as corresponding. Furthermore, in the wake of the clamp, the previously unrecognized alphabetic symmetry between ***a*** and ***z*** has been noticed on account of the increased attention focused on these letters by top-down pressure, resulting in a *first* ⇒ *last* slippage being made. Several other conceptual slippages induced by the active *opposite* concept are also visible in the Trace. Consequently, the final answer description for ***wyz*** includes the themes *AlphabeticPosition: opposite*, *Direction: opposite*,

[2] *GroupType* and *ObjectType* are synonyms for the concepts *GroupCategory* and *ObjectCategory*.

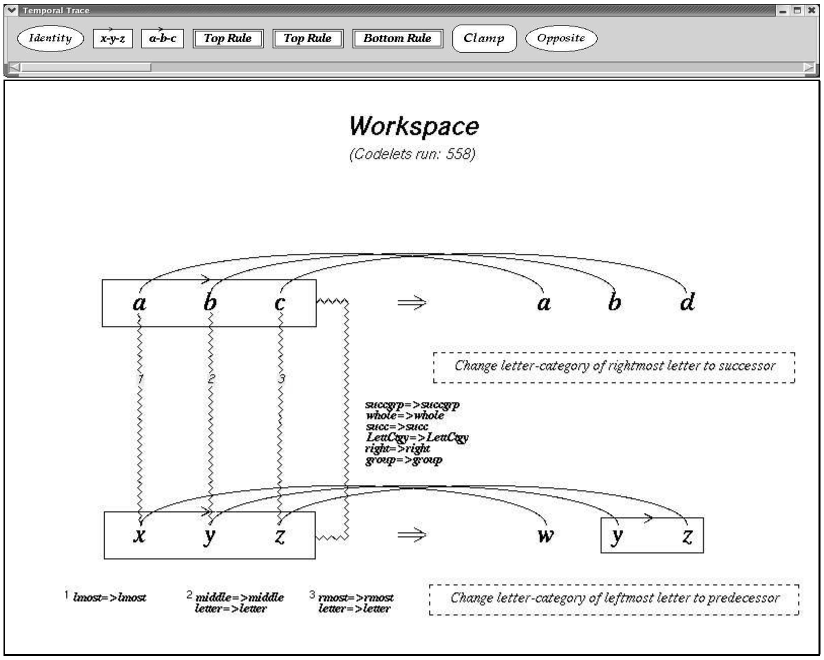


Figure 5: An inconsistent interpretation of the answer ***wyz***

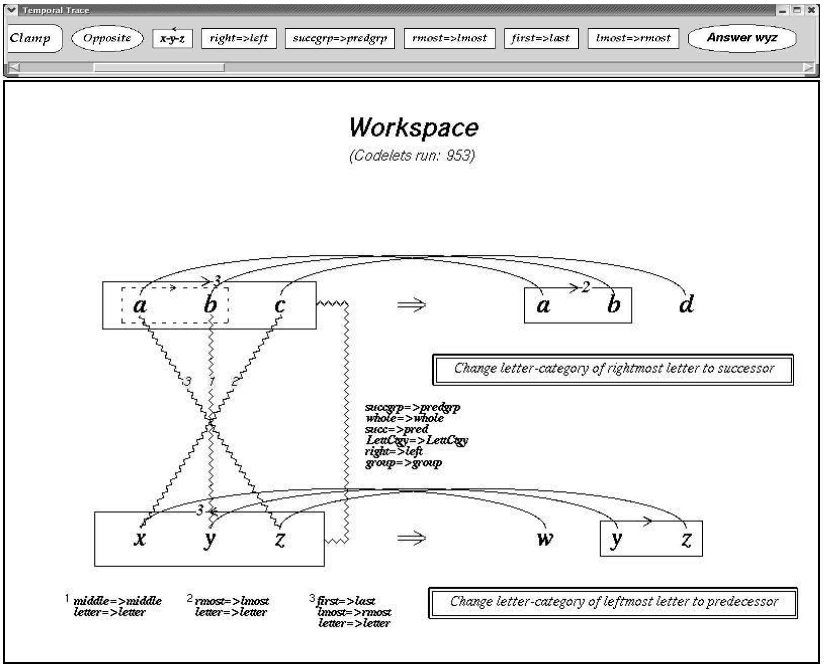


Figure 6: The final, consistent interpretation of ***wyz***

GroupType: opposite, and *StringPosition: opposite*.

6.2 Jootsing

Another type of codelet that watches the action from the high-level vantage point of the Temporal Trace is called a *Jootser* (a term coined by Hofstadter, short for "*j*umping *o*ut *o*f *t*he *s*ystem"). These codelets are responsible for noticing repetitive behavior that the program has fallen into. An example of such behavior arising from repeatedly hitting a snag in ***abc*** ⇒ ***abd; xyz*** ⇒ ***?*** was mentioned earlier. However, *Jootser* codelets are sensitive to other kinds of situations as well. For example, it is possible for Metacat to become "fixated" on some idea, such that it ends up clamping the same pattern over and over again, without making any significant progress. In this case, too, *Jootser* codelets may notice a series of recurring events in the Trace and take action.

For instance, if an analogy problem happens to involve a string that changes in some complicated way, it may be too difficult for the program to build a rule that describes this change. The program may end up repeatedly clamping patterns in a futile effort to spur the creation of new rules. Repetitive clamping behavior can even arise from unsuccessful attempts to break out of a cycle of snags. That is, clamping a pattern in response to a recurring snag may prove to be ineffective, leading only to further snags and more pattern-clamping, rather than to a new interpretation of the problem.

Faced with several similar clamp events in the Trace, a *Jootser* codelet decides probabilistically whether to "joots" based on the number of clamps and the average amount of progress achieved by each. The more clamp events there are, the more likely jootsing is to occur, especially if the amount of progress is low, unless recent clamps appear to be making more headway than earlier ones. Jootsing from repeated clamps, however, does not involve clamping any new patterns in response, in contrast to jootsing from repeated snags. Instead, Metacat simply "gives up" in a graceful manner and stops.

The following two examples illustrate the idea of jootsing. In the first, Metacat tries to justify the answer ***aaabccc*** to the problem ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?***. It eventually gives up after trying unsuccessfully several times to connect the concept of *LetterCategory* in ***eqe*** to the concept of *Length*

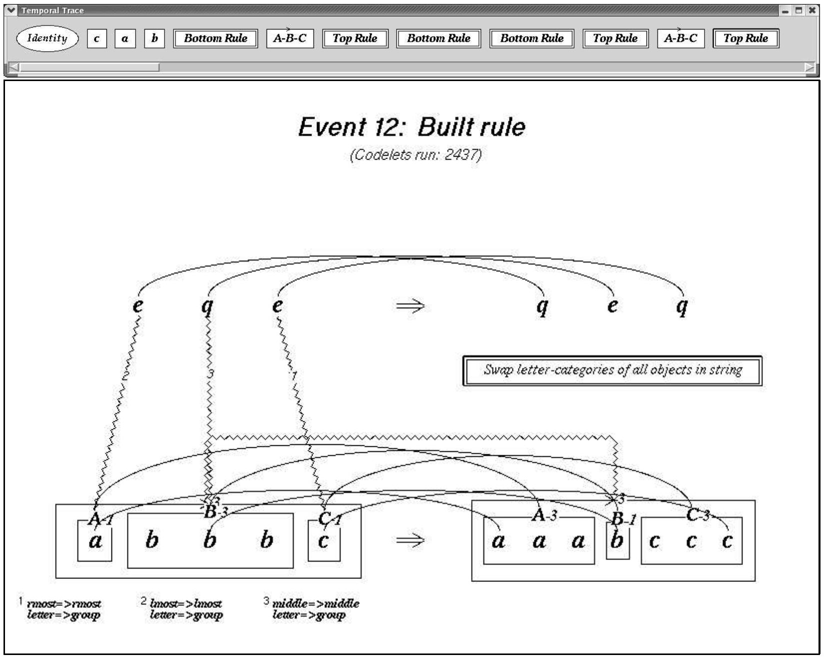
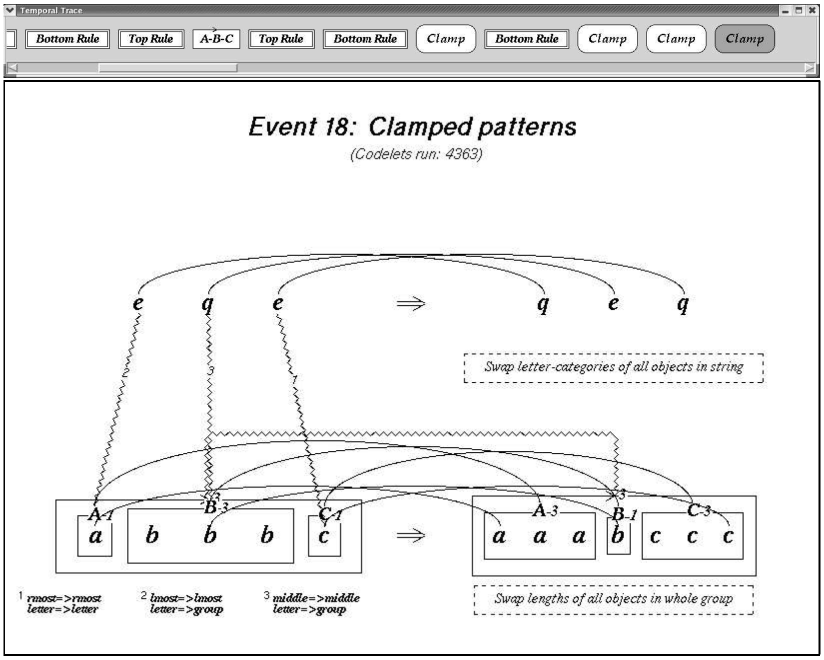


Figure 7: An emerging but still incomplete interpretation of ***aaabccc***

in ***aaabccc***.

By time step 2437, the program has perceived both ***abbbc*** and ***aaabccc*** as successor groups based on the letter-categories *a–b–c*, and has created a variety of top and bottom rules describing ***eqe*** ⇒ ***qeq*** and ***abbbc*** ⇒ ***aaabccc*** (see Fig. 7). Over the next 2000 time steps, the program clamps several patterns of themes as a result of comparing various top and bottom rules to one another. For example, at time step 4363, the program compares a top rule that describes the objects of ***eqe*** as swapping their *letter-categories* with a similar bottom rule that describes the objects of ***abbbc*** as swapping their *lengths*. These rules differ by only one concept, but to make them inter-translatable, a *LetterCategory* ⇒ *Length* slippage must somehow be made between ***eqe*** and ***abbbc***. The program therefore clamps patterns of themes designed to induce the creation of a mapping involving this slippage (see Fig. 8). Unfortunately, however, building such a mapping requires ***eqe*** to be seen as a single, chunked group, which is impossible in the current version of Metacat, since only successorship, predecessorship, or sameness relations are recognized among letters. Thus the program falls into an unsuccessful cycle of pattern-clamping. Eventually, at time step 6196, a *Jootser* codelet notices the series of repeated clamps in the Trace and decides to terminate the run, without having achieved a complete understanding of ***aaabccc*** (see Fig. 9).



Event 18: Clamped patterns

(Codelets run: 4363)

Swap letter-categories of all objects in string

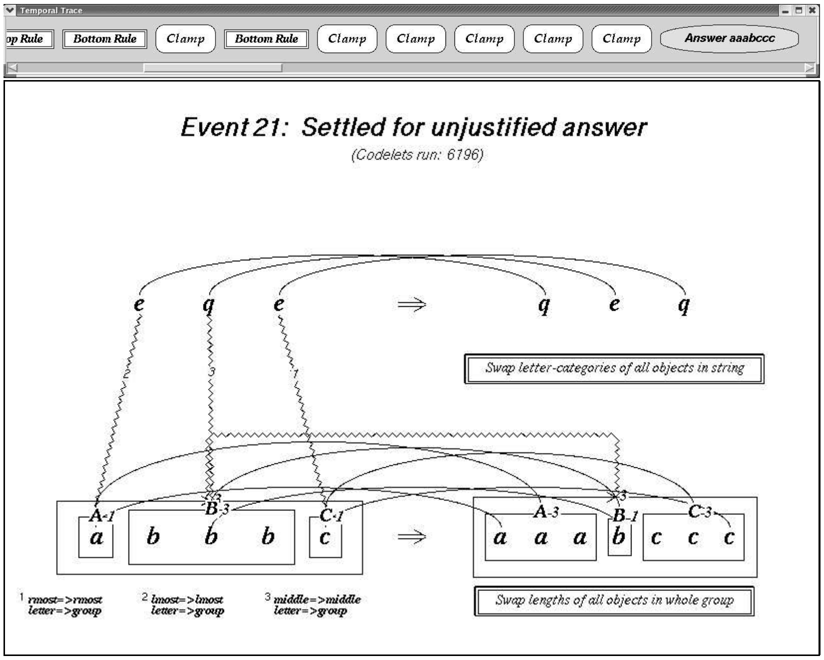
Swap lengths of all objects in whole group

[1] rmost=>rmost letter=>letter

[2] lmost=>lmost letter=>group

[3] middle=>middle letter=>group

Figure 8: Attempting to induce a *LetterCategory* ⇒ *Length* slippage by clamping themes



Event 21: Settled for unjustified answer

(Codelets run: 6196)

Swap letter-categories of all objects in string

Swap lengths of all objects in whole group

[1] rmost=>rmost letter=>group

[2] lmost=>lmost letter=>group

[3] middle=>middle letter=>group

Figure 9: The final, unjustified interpretation of ***aaabccc***

As this example shows, once Metacat recognizes that its attempts to justify an answer are not succeeding, it may decide to settle for an *unjustified answer*, depending on how close it can come to a complete justification. In general, if valid rules exist for describing both the top and bottom string changes, and if these rules are *almost* the same under translation, differing by at most a few concepts, then the program will throw in the towel, reporting its failure to understand how the unjustified slippages arise. It will also include *unjustified themes* based on these slippages in the answer description that is created at the end of the run.

The more unjustified slippages there are, however, the less likely jootsing is to occur. Nevertheless, there is always the possibility that the program will give up too easily, reporting an answer as unjustified when in fact it could be completely justified with further effort, although in practice this does not happen very often. On the other hand, of course, it is impossible for Metacat to know which answers are beyond its ability to justify in principle, since this would require a type of self-knowledge far beyond the capability of the present program (for example, Metacat would have to know that it is *not capable* of seeing ***eqe*** as a single group). In any case, the program at least knows that it has settled for an unjustified answer, and notes this fact, along with the associated unjustified themes, in its Episodic Memory.

The second example of jootsing involves the same problem, ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?***, but this time Metacat must solve it on its own, instead of being given an answer to start with. In this run, the program begins by structuring ***abbbc*** as a successor group composed of the letter ***a***, the group ***bbb***, and the letter ***c***, as in the previous example. The two rules shown below are also created to describe ***eqe*** ⇒ ***qeq***:

- *Swap letter-categories of all objects in string*
- *Change letter-category of leftmost letter to 'q'*
  Change letter-category of middle letter to 'e'
  Change letter-category of rightmost letter to 'q'

Around time step 1100, the program attempts to apply the first rule to ***abbbc***, which results in a snag, since a three-way swap between ***a***, ***b***, and ***c*** is impossible (see Fig. 10). If the second rule had been chosen instead of the first, the program would have found the answer ***qeeeq***, but because this rule is less abstract than the first, it is less likely to be chosen on average.

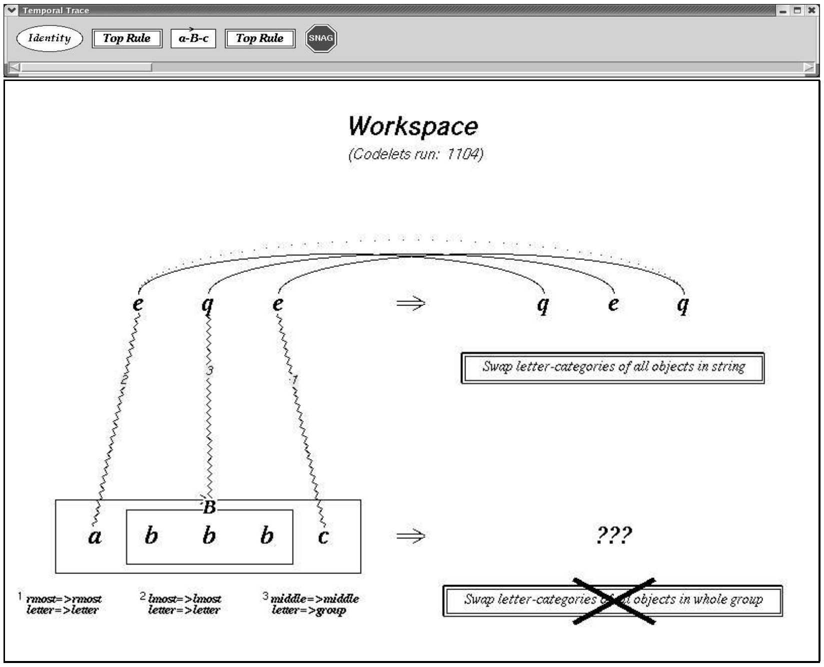

Temporal Trace
Identity
Top Rule
a-B-c
Top Rule
SNAG
Workspace
(Codelets run: 1104)
e q e ⇒ q e q
3
2
1
Swap letter-categories of all objects in string
B
a b b b c ⇒ ???
1 rmost=>rmost letter=>letter
2 lmost=>lmost letter=>letter
3 middle=>middle letter=>group
Swap letter-categories of all objects in whole group

Figure 10: Attempting to swap the components of ***abbbc***

Over the next 3000 time steps, Metacat tries again and again to swap the components of ***abbbc***, often breaking various structures in the process, but always rebuilding them in the same way as before. Eventually, at time step 4280, a *Jootser* codelet notices the pattern of recurring snag events in the Trace, all of which involve the themes *StringPosition: identity*, *ObjectType: identity*, and *ObjectType: different*. These themes arise from the program's interpretation of the letters ***e***, ***q***, and ***e*** in ***eqe*** as corresponding, respectively, to the letter ***a***, the group ***bbb***, and the letter ***c*** in ***abbbc***. The *ObjectType: identity* theme is based on the ***e–a*** and ***e–c*** bridges, while the *ObjectType: different* theme results from the bridge between ***q*** and ***bbb***, since one is a letter and the other is a group.

In an effort to avoid the recurring snag, the codelet probabilistically decides to negatively clamp the *ObjectType: identity* theme. The ensuing thematic pressure results in ***abbbc*** being reinterpreted as a predecessor group going to the left, and a new rule being created to describe ***eqe*** ⇒ ***qeq***, but these new structures do not really change the basic situation. Soon afterwards, another *Jootser* codelet tries again, this time clamping *both ObjectType* themes, which effectively paralyzes the program for the duration of the clamp period, since no structures can be built that are compatible with both of these themes simultaneously. Fig. 11 shows the state of the Workspace and Trace at the time of the latter clamp.

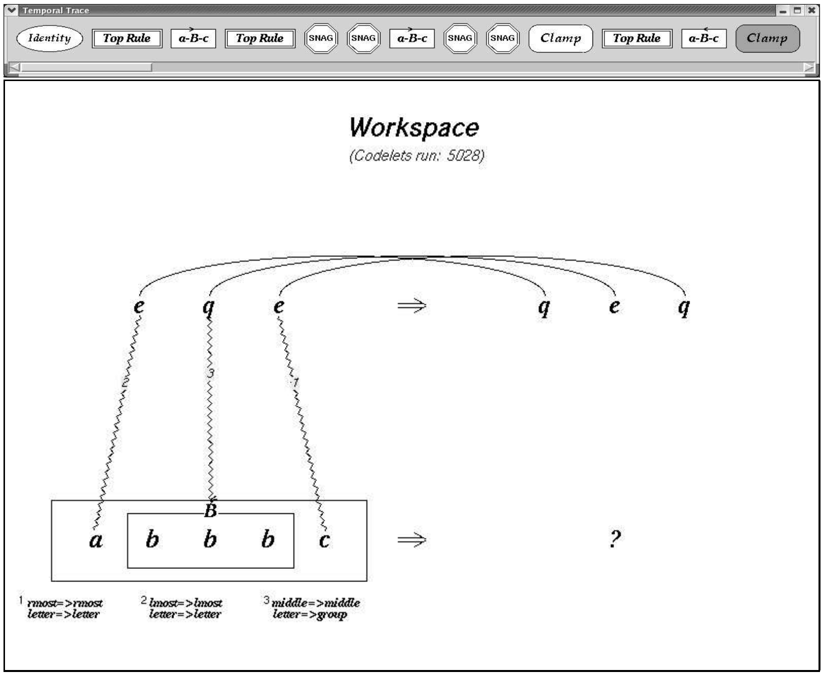


Figure 11: The situation late in the run, after several snags and clamps have occurred

A few hundred codelets later, the program hits the snag again. This is followed shortly thereafter by another clamp. This clamp, like the one before it, achieves no new progress. After hitting the snag yet again, the program finally decides to give up. More precisely, at time step 5933, a *Jootser* codelet notices the three clamp events in the Trace, all of which have overlapping sets of associated themes. Moreover, neither of the two most recent clamps have resulted in any discernible progress, which further increases the probability of jootsing. Consequently, the program prints a termination message and ends the run, instead of just continuing to cycle.

6.3 Levels of Control in Self-Watching Systems

Settling for an unjustified answer after repeatedly trying to make sense of it, as in the first example, or attempting to circumvent a recurring snag by clamping themes, as in the second example, can be thought of as “first-order” jootsing. In contrast, recognizing when repeated attempts to circumvent a snag are leading nowhere, as in the second example, can be thought of as “higher-order” or “meta-level” jootsing—that is, jootsing from repeated attempts at jootsing.

This important distinction can be framed more clearly in terms of event types in the Temporal

Trace. Let us designate as *Type I* an event that occurs directly in response to processing Workspace structures. For example, snag events are of Type I, because they arise from a failed attempt to apply a rule to a string (as shown earlier in Fig. 10). A clamp event that occurs as a result of comparing two rules when trying to justify an answer (as shown in Fig. 8) is also of Type I. Likewise, clamping a pattern of codelet urgencies in an effort to spur the creation of new structures such as rules is a Type I event as well, since this happens in response to poor-quality (or nonexistent) structures in the Workspace. In other words, Type I events in the Trace arise from first-order, *subcognitive* processing activity in the Workspace.

On the other hand, a *Type II* event is one that occurs directly in response to Type I events in the Trace. For example, clamping a pattern of themes in response to a recurring snag (as in the second example of section 6.2) is a Type II event, since it is triggered by noticing a series of snag events in the Trace. In other words, Type II events arise from patterns of activity at the *cognitive* processing level, or, said another way, from viewing subcognitive processing activity *at an appropriately abstract level of description*. Thus first-order jootsing corresponds to noticing a series of Type I events in the Trace that all share similar thematic characterizations, and responding in some appropriate way, while meta-level jootsing corresponds to noticing and responding to Type II events.

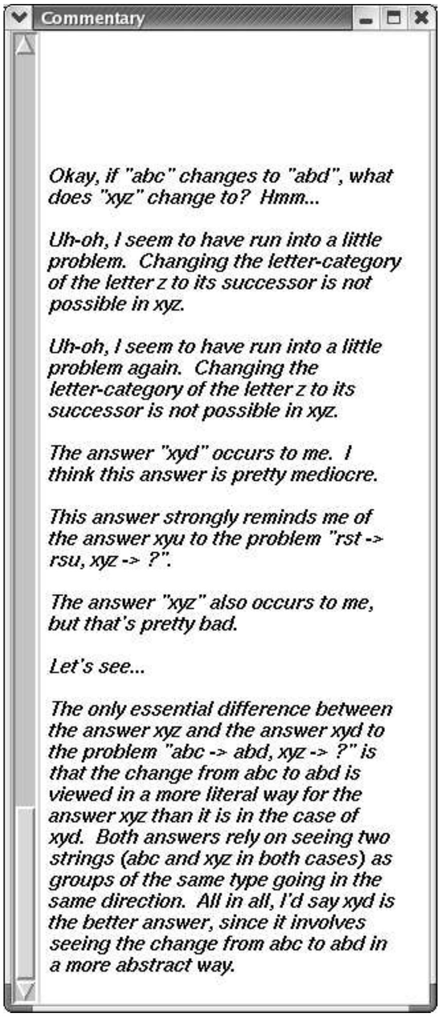
The important point is that the same mechanisms are responsible for both first-order and meta-level jootsing in Metacat—namely, *Jootser* codelets and the explicit representation of processing events in the Temporal Trace. This reflects our belief that a self-watching system should not be organized as a rigid hierarchy of distinct levels, with each level responsible only for detecting and responding to patterns occurring at the level immediately below it, implying the need for an infinite stack of separate "watcher" mechanisms. Instead, a single set of mechanisms should be capable of detecting first-order patterns, higher-order patterns within these patterns, patterns of patterns of patterns, and so on, with all levels fused together and no limit in principle on the potential complexity of the patterns involved (Hofstadter, 1985a).

7 Program-Generated Commentary

As Metacat works on an analogy problem, it displays a running commentary in English summarizing the "ideas" that occur to it as it tries to discover an answer (or to make sense of one provided to it). This narrative, which appears in Metacat's *Commentary* window, corresponds closely to the chain of events in the Temporal Trace, although it is not an event-by-event transcription of the information recorded there. Rather, it consists of explanatory messages generated from time to time by codelets as they go about their business. For example, when Metacat encounters a snag, it reports this fact and briefly explains why the snag has occurred. Upon discovering a new answer, it states its "opinion" of the answer's quality, and mentions any other answers it has seen in the past that the newly-found answer reminds it of. The program also mentions when it is getting "frustrated" by a lack of progress, such as in the case of failing to create good rules for describing string changes. Furthermore, after attempting to focus on some new idea by clamping a pattern of themes, it gives a brief assessment, in retrospect, of the progress achieved by the clamp. The program can also comment on the similarities and differences between various answers, if prompted by the user.

Fig. 12 illustrates the type of commentary typically generated by the program during a run. The example on the left shows a run on the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?*** in which the program hits the *z* snag a couple of times and then answers ***xyd*** (the Temporal Trace from this run was shown earlier in Fig. 4). As it happens, the answer ***xyd*** reminds the program of a similar answer to a different problem that it has already solved. Continuing on, the program then finds the "do-nothing" answer ***xyz***, based on the rule *Change letter-category of letter 'c' to 'd'*. This rule is even more literal-minded than the rule *Change letter-category of rightmost letter to 'd'*. At this point, prompted by the user, the program compares the answer ***xyz*** to the answer ***xyd***, expressing a preference for ***xyd***.

Next, Metacat is given the answer ***dyz*** to the same problem and asked to justify it (Fig. 12, right). In this run, the program has difficulty at first discovering a rule to describe the change from ***xyz*** to ***dyz***. Its comment about "trying harder" arises from clamping a pattern of codelet urgencies in response to this lack of rules. As it turns out, three new rules get created in the wake of this clamp. The program therefore judges the amount of progress made by the clamp as satisfactory. In fact, analyzing the newly-created rules leads the program to subsequently clamp a pattern of themes in



Okay, if "abc" changes to "abd", what does "xyz" change to? Hmm...

Uh-oh, I seem to have run into a little problem. Changing the letter-category of the letter z to its successor is not possible in xyz.

Uh-oh, I seem to have run into a little problem again. Changing the letter-category of the letter z to its successor is not possible in xyz.

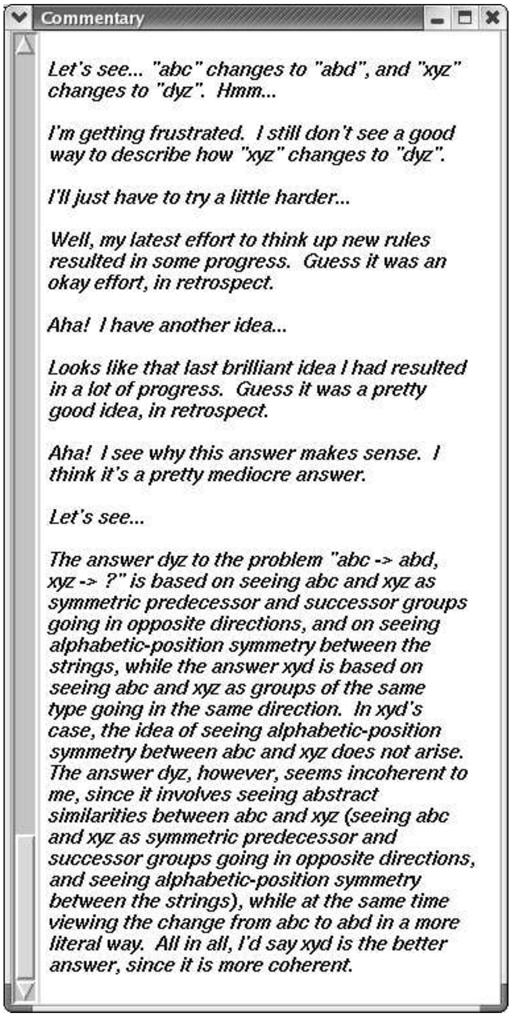
The answer "xyd" occurs to me. I think this answer is pretty mediocre.

This answer strongly reminds me of the answer xyu to the problem "rst -> rsu, xyz -> ?".

The answer "xyz" also occurs to me, but that's pretty bad.

Let's see...

The only essential difference between the answer xyz and the answer xyd to the problem "abc -> abd, xyz -> ?" is that the change from abc to abd is viewed in a more literal way for the answer xyz than it is in the case of xyd. Both answers rely on seeing two strings (abc and xyz in both cases) as groups of the same type going in the same direction. All in all, I'd say xyd is the better answer, since it involves seeing the change from abc to abd in a more abstract way.



Let's see... "abc" changes to "abd", and "xyz" changes to "dyz". Hmm...

I'm getting frustrated. I still don't see a good way to describe how "xyz" changes to "dyz".

I'll just have to try a little harder...

Well, my latest effort to think up new rules resulted in some progress. Guess it was an okay effort, in retrospect.

Aha! I have another idea...

Looks like that last brilliant idea I had resulted in a lot of progress. Guess it was a pretty good idea, in retrospect.

Aha! I see why this answer makes sense. I think it's a pretty mediocre answer.

Let's see...

The answer dyz to the problem "abc -> abd, xyz -> ?" is based on seeing abc and xyz as symmetric predecessor and successor groups going in opposite directions, and on seeing alphabetic-position symmetry between the strings, while the answer xyd is based on seeing abc and xyz as groups of the same type going in the same direction. In xyd's case, the idea of seeing alphabetic-position symmetry between abc and xyz does not arise. The answer dyz, however, seems incoherent to me, since it involves seeing abstract similarities between abc and xyz (seeing abc and xyz as symmetric predecessor and successor groups going in opposite directions, and seeing alphabetic-position symmetry between the strings), while at the same time viewing the change from abc to abd in a more literal way. All in all, I'd say xyd is the better answer, since it is more coherent.

Figure 12: Metacat's commentary from a run on the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?*** in which it found the answers ***xyd*** and ***xyz*** (left), and from a justification run on the same problem in which the program was given the answer ***dyz*** (right).

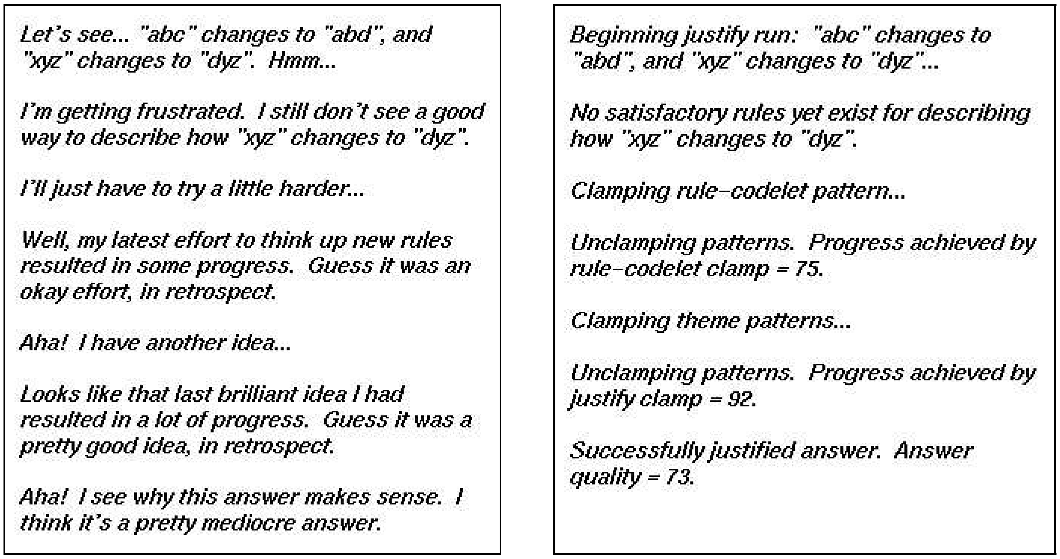
an effort to create a mapping between ***abc*** and ***xyz*** that is compatible with the rules. This second clamp is indicated by the comment, "Aha! I have another idea..." This clamp spurs the creation of many new structures, leading to the interpretation of ***abc*** and ***xyz*** as mirror images of each other, which in turn leads to a successful justification of ***dyz***. The program therefore judges the progress achieved by the second clamp to be very high, even though it considers ***dyz*** itself to be a "pretty mediocre" answer. Finally, again prompted by the user, the program compares this answer to the answer ***xyd*** found earlier, which it judges in the end to be of higher quality than ***dyz***.

From these examples, it may appear that Metacat possesses a sophisticated linguistic ability.

However, it must be stressed that this is not the case. The program's commentary is generated by a set of prefabricated phrase-templates, which get filled in and combined in flexible ways according to context. See Marshall (1999, Chapter 4) for a detailed discussion of these mechanisms. In the run shown in Fig. 12, for example, the explanation of the snag is generated on the basis of the Workspace structures and Slipnet concepts involved in the snag—namely, the letter ***z***, the string ***xyz***, and the concepts of *LetterCategory* and *successor*. As an added touch, the second time the program hits the snag, it inserts the word "again", on account of the fact that a previous snag event exists in the Temporal Trace. In addition, the program uses stock phrases to describe certain numerical values, such as the overall measure of answer quality (*e.g.*, "pretty mediocre", "pretty bad"), or the progress achieved by a clamp (*e.g.*, "some", "a lot of"), or the strength of reminding of one answer by another (*e.g.*, "strongly"). Other phrases are completely canned, such as "I seem to have run into a little problem", which the program prints out whenever it hits a snag, or "Let's see...", which is printed whenever the program compares answers. Furthermore, no type of linguistic *interaction* with the program is possible.

The purpose of Metacat's commentary is to show the progression of activity that occurs during a run in a very user-friendly and somewhat whimsical fashion, as if the program were "thinking out loud" while it solves problems, and also to summarize, in an easily understandable way, the parallels and distinctions between answers that are perceived by the program. It is not intended as a serious model of language processing. As will be discussed below, answers are compared on the basis of their underlying *conceptual representations*, which consist of the themes and rules stored in answer descriptions. Metacat's ability to recognize similarities and differences between analogies *at this representational level* is what counts, not its ability to summarize these comparisons in a human-readable form.

That said, it is important to add that not *all* of the words used by the program are completely devoid of semantic content. To be sure, most of them are: "okay", "think", "mediocre", "I", "me", and so on. However, some of them, such as "letter", "letter-category", "group", "successor", and "direction", denote concepts that the program *does* genuinely understand—in a limited but quite defensible sense—within the confines of its letter-string world. These words correspond to Slipnet



```
Let's see... "abc" changes to "abd", and
"xyz" changes to "dyz". Hmm...

I'm getting frustrated. I still don't see a good
way to describe how "xyz" changes to "dyz".

I'll just have to try a little harder...

Well, my latest effort to think up new rules
resulted in some progress. Guess it was an
okay effort, in retrospect.

Aha! I have another idea...

Looks like that last brilliant idea I had
resulted in a lot of progress. Guess it was a
pretty good idea, in retrospect.

Aha! I see why this answer makes sense. I
think it's a pretty mediocre answer.
```

```
Beginning justify run: "abc" changes to
"abd", and "xyz" changes to "dyz"...

No satisfactory rules yet exist for describing
how "xyz" changes to "dyz".

Clamping rule-codelet pattern...

Unclamping patterns. Progress achieved by
rule-codelet clamp = 75.

Clamping theme patterns...

Unclamping patterns. Progress achieved by
justify clamp = 92.

Successfully justified answer. Answer
quality = 73.
```

Figure 13: Commentary from a justification run with “Eliza mode” on (left) and off (right), showing the one-to-one correspondence between the comments generated in each case.

concepts, whose semantics emerge from the complex ways in which they interact with perceptual processing, as discussed earlier in section 4.

Although the colloquial tone of Metacat’s commentary is meant to be humorous, it raises the potential danger of the so-called “Eliza effect”, which refers to the widespread tendency of people to read far more meaning than is warranted into text generated by a computer program. Clearly, the output generated by Metacat might lead (or mislead) a casual observer into falling for this effect. Therefore, in the interest of transparency, the program can be run in two different linguistic output modes. When running in “Eliza mode”, Metacat generates the type of commentary shown in Fig. 12. With this mode turned off, the program uses more neutral language to describe the events that occur during a run (the explanations generated when comparing answers, however, are not affected). For example, Fig. 13 shows the output from the second run of Fig. 12, alongside the isomorphic output produced with Eliza mode turned off. Exactly the same number of paragraphs are generated in either case.

7.1 Comparing Analogies

When Metacat compares two analogies, it retrieves their answer descriptions from its Episodic Memory and analyzes the themes and rules contained therein. In general, two answer descrip-

Problem/Answer	Themes	Type of Rule
abc ⇒ abd; xyz ⇒ wyz	*AlphabeticPosition:opposite* *StringPosition:opposite*	Abstract
rst ⇒ rsu; xyz ⇒ wyz	*StringPosition:opposite*	Abstract
abc ⇒ abd; xyz ⇒ xyd	*StringPosition:identity*	Literal
rst ⇒ rsu; xyz ⇒ xyu	*StringPosition:identity*	Literal
abc ⇒ abd; xyz ⇒ dyz	*AlphabeticPosition:opposite* *StringPosition:opposite*	Literal
rst ⇒ rsu; xyz ⇒ uyz	*StringPosition:opposite*	Literal

Table 1: Answer descriptions for the ***xyz*** family of analogies

tions may share identical themes (called *common* themes), they may have themes of the same type which differ by relation (called *differing* themes), or one or both answers may have themes that are not present in the other answer at all (called *unique* themes). For example, consider again the ***xyz*** family of analogies discussed in section 3 (Fig. 1). Table 1 shows some of the information stored in the answer descriptions created by Metacat for these analogies, including themes characterizing the mapping between the initial string and target string. (For clarity, not all of the stored information is shown.) The answers ***xyd*** and ***xyu*** share a common *StringPosition:identity* theme. On the other hand, ***xyu*** and ***uyz*** are based on the differing themes of *StringPosition:identity* and *StringPosition:opposite*. In the case of the two ***wyz*** answers, the first one contains a unique *AlphabeticPosition:opposite* theme.

Analyzing the themes and rules shown in Table 1 brings out clearly the similarities and differences between these analogies. For example, a crucial distinction between the first ***wyz*** answer and ***dyz*** is the abstractness of the rule used to describe ***abc*** ⇒ ***abd***. The descriptions of ***xyd*** and ***xyu*** are identical, reflecting the strong underlying similarity of these two literal-minded analogies. The difference between the two ***wyz*** analogies lies in the presence or absence of the idea of alphabetic symmetry. Moreover, the way in which these analogies differ is precisely the same as the way in which ***dyz*** differs from ***uyz***.

The coherence of an answer can be judged by comparing the abstractness of the answer's themes with the abstractness of the concepts making up the answer's rule. For example, ***dyz*** is characterized by themes involving the abstract concept of *opposite*, but depends on a literal-minded interpretation of ***abc*** ⇒ ***abd***. This "dissonance" is the reason that Metacat considers ***dyz*** to be an incoherent analogy, as it explained in Fig. 12.

The following is a sampling of Metacat's explanations of the similarities and differences between some of the analogies in Table 1. To generate these explanations, the program was first run (in justify mode) on each of the answers, and was then asked to compare them. The figures show the output generated by the program.

In Fig. 14, the program compares the answers ***wyz*** and ***xyd*** to the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, and explains why it considers ***wyz*** to be the better analogy. The phrase "a richer set of ideas" refers to the fact that ***wyz***'s answer description contains more themes than ***xyd***'s description.

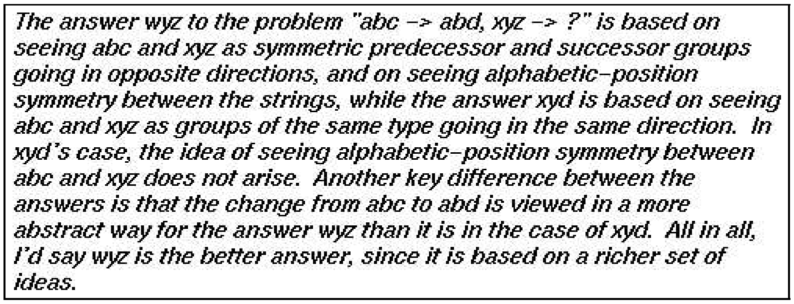
The next examples illustrate answer comparison across different problems, namely, ***abc*** ⇒ ***abd; xyz*** ⇒ ***?*** and ***rst*** ⇒ ***rsu; xyz*** ⇒ ***?***. This amounts to comparing answers "vertically" in Fig. 1. In Fig. 15, the program explains why it considers the answers ***xyd*** and ***xyu*** to be fundamentally the same analogy. As the program notes, the rules giving rise to these answers are very similar, since they both involve changing the rightmost letter in a literal-minded way. The program assigns a rating of "pretty mediocre" to each answer, based on the low degree of abstractness of the answers' underlying themes and rules.

In Fig. 16, the two ***wyz*** answers are compared. In this case, the program recognizes the essential difference between these analogies—namely, the presence of alphabetic symmetry in one but not the other—despite the superficial identity of the two answers.

In Fig. 17, the program compares the answers ***dyz*** and ***uyz***, each of which involves a somewhat incoherent blend of abstract and literal-minded perspectives. As in the previous ***wyz*** vs. ***wyz*** case, the program identifies the presence or absence of alphabetic symmetry as the fundamental difference between these two analogies. It also notes their peculiar incoherence, expressing a preference for ***uyz***.

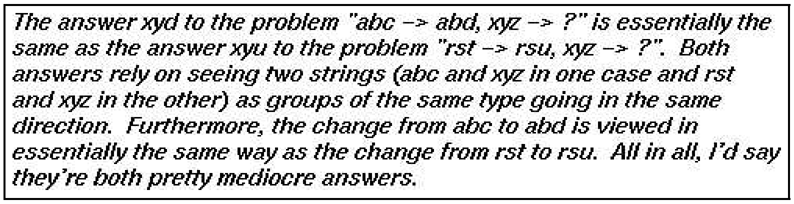
7.2 Reminding

Closely related to answer comparison is the phenomenon of reminding, in which one answer may trigger the retrieval from memory of other answers that are in some way similar. This may happen



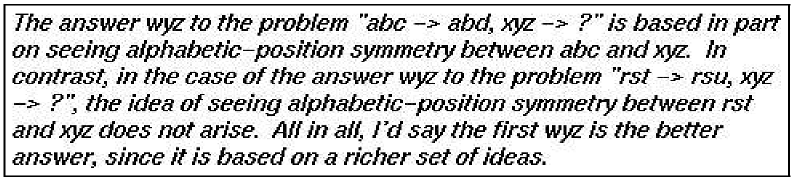
The answer wyz to the problem "abc -> abd, xyz -> ?" is based on seeing abc and xyz as symmetric predecessor and successor groups going in opposite directions, and on seeing alphabetic-position symmetry between the strings, while the answer xyd is based on seeing abc and xyz as groups of the same type going in the same direction. In xyd's case, the idea of seeing alphabetic-position symmetry between abc and xyz does not arise. Another key difference between the answers is that the change from abc to abd is viewed in a more abstract way for the answer wyz than it is in the case of xyd. All in all, I'd say wyz is the better answer, since it is based on a richer set of ideas.

Figure 14: ***abc ⇒ abd; xyz ⇒ xyd*** versus ***abc ⇒ abd; xyz ⇒ wyz***



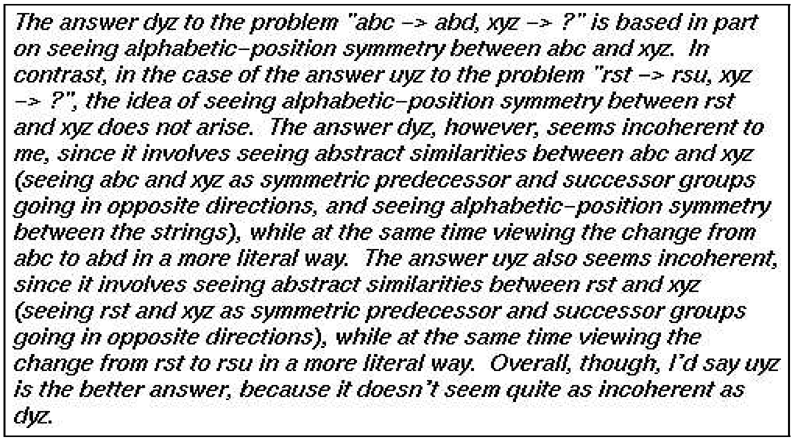
The answer xyd to the problem "abc -> abd, xyz -> ?" is essentially the same as the answer xyu to the problem "rst -> rsu, xyz -> ?". Both answers rely on seeing two strings (abc and xyz in one case and rst and xyz in the other) as groups of the same type going in the same direction. Furthermore, the change from abc to abd is viewed in essentially the same way as the change from rst to rsu. All in all, I'd say they're both pretty mediocre answers.

Figure 15: ***abc ⇒ abd; xyz ⇒ xyd*** versus ***rst ⇒ rsu; xyz ⇒ xyu***



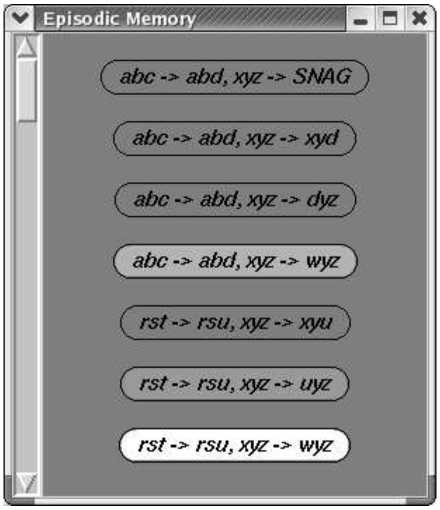
The answer wyz to the problem "abc -> abd, xyz -> ?" is based in part on seeing alphabetic-position symmetry between abc and xyz. In contrast, in the case of the answer wyz to the problem "rst -> rsu, xyz -> ?", the idea of seeing alphabetic-position symmetry between rst and xyz does not arise. All in all, I'd say the first wyz is the better answer, since it is based on a richer set of ideas.

Figure 16: ***abc ⇒ abd; xyz ⇒ wyz*** versus ***rst ⇒ rsu; xyz ⇒ wyz***



The answer dyz to the problem "abc -> abd, xyz -> ?" is based in part on seeing alphabetic-position symmetry between abc and xyz. In contrast, in the case of the answer uyz to the problem "rst -> rsu, xyz -> ?", the idea of seeing alphabetic-position symmetry between rst and xyz does not arise. The answer dyz, however, seems incoherent to me, since it involves seeing abstract similarities between abc and xyz (seeing abc and xyz as symmetric predecessor and successor groups going in opposite directions, and seeing alphabetic-position symmetry between the strings), while at the same time viewing the change from abc to abd in a more literal way. The answer uyz also seems incoherent, since it involves seeing abstract similarities between rst and xyz (seeing rst and xyz as symmetric predecessor and successor groups going in opposite directions), while at the same time viewing the change from rst to rsu in a more literal way. Overall, though, I'd say uyz is the better answer, because it doesn't seem quite as incoherent as dyz.

Figure 17: ***abc ⇒ abd; xyz ⇒ dyz*** versus ***rst ⇒ rsu; xyz ⇒ uyz***



abc -> abd, xyz -> SNAG

abc -> abd, xyz -> xyd

abc -> abd, xyz -> dyz

abc -> abd, xyz -> wyz

rst -> rsu, xyz -> xyu

rst -> rsu, xyz -> uyz

rst -> rsu, xyz -> wyz

Figure 18: Six answer descriptions and one snag description in memory

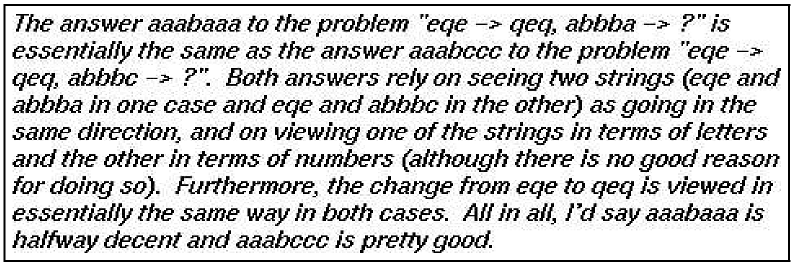
whenever a new answer is discovered or justified by the program. When a new answer is found, the answer description created from the information in the Temporal Trace acts as an index into memory, causing other stored answer descriptions to become activated in proportion to their similarity to the new answer. Similarity between answer descriptions is determined by a numerical measure from 0 to 100 called the *distance*, which measures the amount of overlap of the answer descriptions' themes and concepts. If the activation level of an answer description exceeds a fixed threshold, Metacat will be reminded of the answer, with the activation level corresponding to the strength of recall.

Fig. 18 shows an example of the state of Metacat's memory upon discovering the answer ***wyz*** to the problem ***rst*** ⇒ ***rsu; xyz*** ⇒ ***?***, after having seen a few other answers to this problem and to the problem ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***. There is also a snag description for ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***, indicating that the program ran into a snag when solving this problem on its own. The activation levels of answers are indicated by shades of grey, ranging from white for fully-activated answers to dark grey for dormant ones (so that the less strongly activated an answer is, the more it appears to fade into the background of Metacat's memory). In this example, ***wyz*** is the most active answer, since it was just found. It has partially activated the other ***wyz*** answer, and, to a lesser extent, ***uyz***. The other answers, however, are too distant from ***wyz*** to be recalled. As a result, Metacat reports in its Commentary window that the newly-found answer "somewhat" reminds it of the other ***wyz*** answer, and "vaguely" reminds it of ***uyz***. (The program uses the terms "vaguely", "somewhat", and "strongly" to describe the activation levels of answer descriptions, corresponding respectively to the numerical ranges 1–30, 31–70, and 71–100.)

Snag descriptions enable Metacat to "appreciate" certain answers in ways that otherwise would not be possible. For example, consider again the answer ***aaabccc*** to the problem ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?***. As we saw in the sample runs of section 6.2, the program is unable to get this answer on its own, because it is incapable of perceiving *e–q–e* and *1–3–1* as unified chunks, due to the absence of predecessor, successor, or sameness relations among the parts. Consequently, it is unable to connect the idea of *letter* to the idea of *number* at an abstract level, and therefore never sees these ideas as playing analogous roles in ***eqe*** and ***abbbc***. Instead, it ends up repeatedly attempting to swap the ***a***'s, ***b***'s, and ***c***'s. On the other hand, if this answer is provided by the user, the program can make sense of it, although in an incomplete way. It still considers the connection between *letter* and *number* to be an "unjustified idea". More precisely, it includes an unjustified theme in the answer description for ***aaabccc*** based on its failure to make the slippage *LetterCategory* ⇒ *Length*.

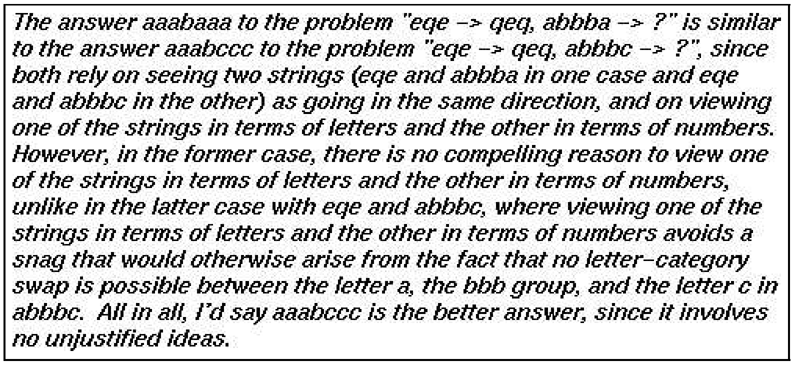
The same is true for the answer ***aaabaaa*** to the related problem ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***?***. Metacat can almost make sense of it, but cannot get it on its own. However, there is a crucial difference between ***aaabaaa*** and ***aaabccc***, as was pointed out earlier in section 3. In ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***?***, swapping letter-categories is perfectly feasible, so there is no need to view ***abbba*** as *1–3–1*. That is, no snag arises in this problem. In a sense, then, the answer ***aaabccc*** is the better analogy, since seeing ***abbbc*** as *1–3–1* provides an elegant way around a snag, while seeing ***abbba*** as *1–3–1* is unnecessary. Metacat can make this observation, but it can only do so if it knows that the problem ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?*** leads to a snag. If it has tried this problem on its own, it will know this, because a corresponding snag description will exist in memory. Conversely, if it is shown the answer ***aaabccc*** without having first tried to solve the problem itself, it will remain unaware of the possibility of a snag arising, and will not perceive this subtle distinction between the two analogies.

The following experiment illustrates this behavior. First, Metacat's memory was cleared in order to reset the program to a "tabula rasa" state. It was then shown the analogy ***eqe*** ⇒ ***qeq; abbba*** ⇒ ***aaabaaa*** and asked to justify it. At the end of the run, the program created an answer



The answer aaabaaa to the problem "eqe -> qeq, abbba -> ?" is essentially the same as the answer aaabccc to the problem "eqe -> qeq, abbbc -> ?". Both answers rely on seeing two strings (eqe and abbba in one case and eqe and abbbc in the other) as going in the same direction, and on viewing one of the strings in terms of letters and the other in terms of numbers (although there is no good reason for doing so). Furthermore, the change from eqe to qeq is viewed in essentially the same way in both cases. All in all, I'd say aaabaaa is halfway decent and aaabccc is pretty good.

Figure 19: ***aaabaaa*** versus ***aaabccc*** before encountering the snag



The answer aaabaaa to the problem "eqe -> qeq, abbba -> ?" is similar to the answer aaabccc to the problem "eqe -> qeq, abbbc -> ?", since both rely on seeing two strings (eqe and abbba in one case and eqe and abbbc in the other) as going in the same direction, and on viewing one of the strings in terms of letters and the other in terms of numbers. However, in the former case, there is no compelling reason to view one of the strings in terms of letters and the other in terms of numbers, unlike in the latter case with eqe and abbbc, where viewing one of the strings in terms of letters and the other in terms of numbers avoids a snag that would otherwise arise from the fact that no letter-category swap is possible between the letter a, the bbb group, and the letter c in abbbc. All in all, I'd say aaabccc is the better answer, since it involves no unjustified ideas.

Figure 20: ***aaabaaa*** versus ***aaabccc*** after encountering the snag

description for ***aaabaaa***, which it then stored in memory. Next, the program was shown the analogy ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***aaabccc***. At the end of the second run, the program reported that ***aaabccc*** strongly reminded it of the first answer, ***aaabaaa*** (reminding strength: 80). At this point, the program had not yet attempted to solve ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?*** on its own, and therefore did not know that a snag can arise. When asked to compare these two analogies, the program reported that it saw essentially no differences between them. Fig. 19 shows the program's commentary.

The program was then reset to a tabula rasa state and asked to justify ***aaabaaa***, just as before. However, it was next given the problem ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?*** to work on its own, with no answer provided. In this run, the program attempted unsuccessfully to swap the letters of ***abbbc*** a couple of times, and then happened to discover the more literal-minded answer ***qeeeq***. The failure to swap the letters, however, caused a snag description to be created for this problem in memory. Next, the program was shown the answer ***aaabccc*** to ***eqe*** ⇒ ***qeq; abbbc*** ⇒ ***?***, as before, and asked to justify it. This time, the program reported that ***aaabccc*** reminded it only vaguely of ***aaabaaa*** (reminding strength: 20), indicating that it perceived the analogies as being quite different—although still recognizably related. The program's commentary is shown in Fig. 20.

8 Discussion

A number of researchers have developed cognitive models that incorporate architectural principles similar to those of Metacat, including emergent processing arising from many nondeterministic agents acting concurrently, and the spreading of activation among nodes of a semantic network in response to context-sensitive pressures. Kokinov's DUAL cognitive architecture, which forms the basis of the AMBR1 and AMBR2 models of analogical reasoning and memory retrieval developed by Kokinov and Petrov, is a case in point (Kokinov, 1994a, 1994b; Kokinov & Petrov, 2001). The development of these models has been guided by the belief that subprocesses underlying analogy-making should be integrated into a larger cognitive system comprising perception, memory, learning, and reasoning. As in Metacat, dynamic context-sensitive emergent processing plays a central role in DUAL and AMBR, allowing for the close interaction of representation-building, mapping, transfer, and reminding.

Despite their architectural similarities, however, Metacat and AMBR differ in terms of the relative emphasis each model places on different aspects of cognition. The more recent AMBR2 model (Kokinov & Petrov, 2001) is particularly strong in its approach to modeling the storage and retrieval of memory episodes. In AMBR2, episode representations are highly emergent, decentralized, and context-sensitive, and interact with the mapping process in a psychologically plausible manner. In contrast, Metacat currently lacks sophisticated mechanisms for episodic memory indexing and retrieval. In the current version of the program, when a new answer is discovered, the newly-created answer description is individually compared to all others stored in memory, in order to determine the new activation levels of the stored descriptions—and hence which answers will be recalled as a result of finding the new answer. This simplistic approach does not scale well if many answers exist in memory, and is thus unsatisfactory in principle. Furthermore, as Kokinov and Petrov have pointed out, these memory structures are essentially localized and static (although their activation levels may change, as mentioned above). Unlike themes and Slipnet concepts, answer descriptions do not interact with each other through spreading activation, and do not actively influence perceptual processing, as they undoubtedly should in order to model priming effects and other influences of previous problem-solving experiences on perception. Developing better mechanisms for episodic memory organization and retrieval in Metacat is thus a high priority for future research.

On the other hand, Metacat is strongly committed to modeling concepts as active, dynamic entities that acquire their meanings from within the system itself, through their interactions with perception, as discussed earlier in section 1. AMBR also models concepts, but their meanings are not tied to the system's own perceptions in the same way as in Metacat. For example, AMBR may solve analogy problems involving the concepts of *water* and *teapot*, but the structures representing these concepts in memory presumably do not become activated by the system's perception of real water or real teapots. In contrast, the concepts behind Metacat's analogies, such as *letter* or *successor-group*, acquire their meanings precisely as a consequence of how they respond to the perception of "real" letters and groups in Metacat's microworld.

Another important difference is Metacat's focus on modeling *self-perception*, an aspect of cognition that is not addressed by most other models of analogy-making. As we saw earlier, the information gleaned from self-watching plays a crucial role in the high-level characterization of answers, enabling the program to perceive abstract similarities and differences between analogies as a whole. We believe that a psychologically realistic and complete model of analogy-making should offer some account of higher perceptual levels, including those that reflect aspects of the system's own behavior. In our model, the mechanisms responsible for internal self-perception are not fundamentally different from those responsible for external perception. Both involve the building and manipulation of structures by codelets, whether in the Temporal Trace (for internal perception) or in the Workspace (for external perception). Furthermore, these processes are tightly interwoven, and are highly dependent on the context-sensitive activations of concepts.

Metacat also shares similarities with case-based reasoning (CBR) approaches to analogy (Kolodner, 1993, 1994; Leake, 1996). For instance, Metacat's stored answer descriptions can be likened to cases in CBR, since they form a corpus of experience on which the program can draw when faced with new situations. The discovery of a new answer may trigger the retrieval of similar answers that the program has seen in the past, in a way reminiscent of the retrieval of stored cases in CBR according to their degree of similarity to the target situation. In Metacat, retrieved answers are compared to the current answer by analyzing the similarities and differences between the answers' associated themes. This is roughly akin to comparing cases in CBR in order to determine which aspects of a retrieved case can be applied directly to the target situation without modification, and which aspects must be adapted to fit it. Finally, Metacat's snag descriptions can be viewed as cases that store failure information about analogies.

However, there are important differences between CBR and Metacat. First of all, even though Metacat solves analogy problems, it was not conceived as a model of problem-solving *per se*. Rather, its focus is on modeling the way in which context-sensitive concepts allow analogies to be perceived and understood. It is more concerned with analogical *perception* (and self-perception), than with analogical *reasoning* employed specifically as a tool for solving problems. Moreover, the goal of much CBR work has been to create systems that learn from experience to solve problems in an increasingly effective or efficient manner, whereas in Metacat the notion of improving the program's performance on analogy problems is not relevant. However, some recent CBR-based approaches to modeling creativity (Bento & Cardoso, 2001; Cardoso & Wiggins, 2002) seem to be more in harmony with Metacat's goals than previous CBR systems have been.

9 Conclusion

A prime objective of this research is to explore how adaptable, context-sensitive concepts can give rise to understanding by enabling analogies between apparently dissimilar situations to be perceived. The present work extends and deepens the ideas developed in Copycat by incorporating mechanisms for self-watching, episodic memory, and reminding into the model. These mechanisms make it possible for Metacat to compare and contrast analogies in an insightful way. The ability of the program to perceive subtle parallels and distinctions between analogies represents a significant step beyond the perceptual abilities of Copycat, although much work still remains to be done.

The examples presented in section 7 illustrate Metacat's ability to observe and describe its own behavior, to recall previously-encountered answers, and to explain the similarities and differences it perceives between analogies. This ability relies on storing abstract descriptions of answers and processing events, characterized by patterns of themes, in memory. It is important to emphasize that answer descriptions are just organized collections of Slipnet concepts, since they are composed of themes and rules, which are in turn composed of concepts. These concepts, as the fundamental building-blocks of answer descriptions, form the substrate on which the program's understanding of analogies is based, and acquire their semantics through the ways in which they respond to situations in Metacat's letter-string world. Consequently, the English-language commentary generated by the program about analogies, although just a surface-level veneer in many ways, ultimately rests on a deeper foundation of conceptual representation tied to perception.

Appendix: Source Code

The complete source code for Metacat is available, along with instructions for downloading and running the program, at **http://www.cogsci.indiana.edu/metacat**. Demos of the examples discussed in this paper and in Marshall (1999) are included with the program.

References

Bento, C., & Cardoso, A. (Eds.). (2001). *Proceedings of the Workshop on Creative Systems: Approaches to Creativity in AI and Cognitive Science*, ICCBR 2001, Vancouver, Canada.

Blank, D., Meeden, L., & Marshall, J. (1992). Exploring the symbolic/subsymbolic continuum: A case study of RAAM. In Dinsmore, J. (Ed.), *The Symbolic and Connectionist Paradigms: Closing the Gap*, pp. 113–148. Lawrence Erlbaum Associates, Hillsdale, NJ.

Blank, D. S. (1997). *Learning to See Analogies: A Connectionist Exploration*. Ph.D. thesis, Indiana University, Bloomington, IN.

Carbonell, J. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Michalski, R., Carbonell, J., & Mitchell, T. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Volume 2*, pp. 371–392. Morgan Kaufmann, San Francisco.

Cardoso, A., & Wiggins, G. (Eds.). (2002). *Proceedings of the AISB'02 Symposium on AI and Creativity in Arts and Science*, London, England.

Chalmers, D. J. (1990). Syntactic transformations on distributed representations. *Connection Science*, *2*, 53–62.

Chalmers, D. J., French, R. M., & Hofstadter, D. R. (1992). High-level perception, representation, and analogy: A critique of artificial intelligence methodology. *Journal of Experimental and Theoretical Artificial Intelligence*, *4*(3), 185–211.

Chi, M., Bassok, M., Lewis, M., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, *13*, 145–182.

Chi, M. T. H., de Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, *18*, 439–477.

Chrisman, L. (1991). Learning recursive distributed representations for holistic computation. *Connection Science*, *3*(4), 345–366.

Eliasmith, C., & Thagard, P. (2001). Integrating structure and meaning: a distributed model of analogical mapping. *Cognitive Science*, *25*, 245–286.

Eskridge, T. C. (1994). A hybrid model of continuous analogical reasoning. In Holyoak, K. J., & Barnden, J. A. (Eds.), *Advances in Connectionist and Neural Computation Theory, Volume 2: Analogical Connections*, pp. 207–246. Ablex, Norwood, NJ.

Falkenhainer, B., Forbus, K. D., & Gentner, D. (1990). The structure-mapping engine. *Artificial Intelligence*, *41*(1), 1–63.

Forbus, K. D., Ferguson, R. W., & Gentner, D. (1994). Incremental structure-mapping. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pp. 313–318. Lawrence Erlbaum Associates.

Forbus, K. D., Gentner, D., & Law, K. (1995). MAC/FAC: A model of similarity-based retrieval. *Cognitive Science*, *19*, 141–205.

Forbus, K. D., Gentner, D., Markman, A. B., & Ferguson, R. W. (1998). Analogy just looks like high-level perception: Why a domain-general approach to analogical mapping is right. *Journal of Experimental and Theoretical Artificial Intelligence*, *10*(2), 231–257.

French, R. M. (1995). *The Subtlety of Sameness: A Theory and Computer Model of Analogy-Making*. MIT Press/Bradford Books, Cambridge, MA.

French, R. M. (2002). The computational modeling of analogy-making. *Trends in Cognitive Sciences*, *6*(5), 200–205.

Gasser, M. (1993). The structure grounding problem. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pp. 149–152. Lawrence Erlbaum Associates.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, *7*(2), 155–170.

Goldstone, R., Medin, D., & Gentner, D. (1991). Relational similarity and the nonindependence of features in similarity judgments. *Cognitive Psychology*, *23*, 222–262.

Halford, G. S., Wilson, W. H., Guo, J., Gayler, R. W., Wiles, J., & Stewart, J. E. M. (1994). Connectionist implications for processing capacity limitations in analogies. In Holyoak, K. J., & Barnden, J. A. (Eds.), *Advances in Connectionist and Neural Computation Theory, Volume 2: Analogical Connections*, pp. 363–415. Ablex, Norwood, NJ.

Harnad, S. (1990). The symbol grounding problem. *Physica D*, *42*, 335–346.

Hofstadter, D. R. (1984). The Copycat project: An experiment in nondeterminism and creative analogies. AI Memo 755, MIT Artificial Intelligence Laboratory.

Hofstadter, D. R. (1985a). On the seeming paradox of mechanizing creativity. In *Metamagical Themas*, chap. 23, pp. 526–546. Basic Books, New York.

Hofstadter, D. R. (1985b). Waking up from the Boolean dream: Subcognition as computation. In *Metamagical Themas*, chap. 26, pp. 631–665. Basic Books, New York.

Hofstadter, D. R., & FARG (1995). *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. Basic Books, New York. Co-authored with members of the Fluid Analogies Research Group.

Holyoak, K. J., & Hummel, J. E. (2001). Toward an understanding of analogy within a biological symbol system. In Gentner, D., Holyoak, K., & Kokinov, B. (Eds.), *The Analogical Mind: Perspectives from Cognitive Science*, pp. 161–195. MIT Press, Cambridge, MA.

Holyoak, K. J., & Thagard, P. (1989). Analogical mapping by constraint satisfaction. *Cognitive Science*, *13*(3), 295–355.

Holyoak, K. J., & Thagard, P. (1995). *Mental Leaps: Analogy in Creative Thought*. MIT Press/Bradford Books, Cambridge, MA.

Hummel, J. E., & Holyoak, K. J. (1996). LISA: A computational model of analogical inference and schema induction. In *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society*, pp. 352–357. Lawrence Erlbaum Associates.

Hummel, J. E., & Holyoak, K. J. (1997). Distributed representations of structure: A theory of analogical access and mapping. *Psychological Review*, *104*, 427–466.

Kanerva, P. (1996). Binary spatter-coding of ordered K-tuples. In von der Malsburg, C., von Seelen, W., Vorbrüggen, J. C., & Sendhoff, B. (Eds.), *Artificial Neural Networks – ICANN '96 Proceedings, Bochum, Germany*, pp. 869–873, Berlin. Springer.

Kanerva, P. (1998). Dual role of analogy in the design of a cognitive computer. In Holyoak, K., Gentner, D., & Kokinov, B. (Eds.), *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, pp. 164–170. New Bulgarian University, Sofia, Bulgaria.

Kokinov, B. N. (1994a). The context-sensitive cognitive architecture DUAL. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pp. 502–507. Lawrence Erlbaum Associates.

Kokinov, B. N. (1994b). A hybrid model of reasoning by analogy. In Holyoak, K. J., & Barnden, J. A. (Eds.), *Advances in Connectionist and Neural Computation Theory, Volume 2: Analogical Connections*, pp. 247–318. Ablex, Norwood, NJ.

Kokinov, B. N., & Petrov, A. (2001). Integrating memory and reasoning in analogy-making: The AMBR model. In Gentner, D., Holyoak, K., & Kokinov, B. (Eds.), *The Analogical Mind: Perspectives from Cognitive Science*, pp. 59–124. MIT Press, Cambridge, MA.

Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann, San Francisco.

Kolodner, J. (1994). Understanding creativity: A case-based approach. In Richter, M. M., Wess, S., Althoff, K. D., & Maurer, T. (Eds.), *Topics in Case-Based Reasoning, selected papers from the First European Workshop on Case-Based Reasoning*. Springer-Verlag.

Leake, D. B. (Ed.). (1996). *Case-Based Reasoning: Experiences, Lessons, & Future Directions*. MIT Press/AAAI Press, Cambridge, MA.

Marshall, J. B. (1999). *Metacat: A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception*. Ph.D. thesis, Indiana University, Bloomington, IN.

Marshall, J. B., & Hofstadter, D. R. (1997). The Metacat project: A self-watching model of analogy-making. *Cognitive Studies: Bulletin of the Japanese Cognitive Science Society*, *4*(4), 57–71.

Mitchell, M. (1993). *Analogy-making as Perception*. MIT Press/Bradford Books, Cambridge, MA.

Plate, T. A. (1994). *Distributed representations and nested compositional structure*. Ph.D. thesis, University of Toronto, Canada.

Plate, T. A. (1998). Structured operations with distributed vector representations. In Holyoak, K., Gentner, D., & Kokinov, B. (Eds.), *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, pp. 154–163. New Bulgarian University, Sofia, Bulgaria.

Smith, E. E., & Medin, D. L. (1981). *Categories and Concepts*. Harvard University Press, Cambridge, MA.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, *46*, 159–216.

Thagard, P., Holyoak, K., Nelson, G., & Gochfield, D. (1990). Analog retrieval by constraint satisfaction. *Artificial Intelligence*, *46*(3), 259–310.

Tversky, A. (1977). Features of similarity. *Psychological Review*, *84*, 327–52.

Veloso, M. M., & Carbonell, J. G. (1993). Derivational analogy in PRODIGY: Automating case acquisition, storage and utilisation. *Machine Learning*, *10*, 249–278.

Veloso, M. (1994). *Planning and Learning by Analogical Reasoning*. Springer-Verlag, Berlin.

Wilson, W. H., Halford, G. S., Gray, B., & Phillips, S. (2001). The STAR-2 model for mapping hierarchically structured analogs. In Gentner, D., Holyoak, K., & Kokinov, B. (Eds.), *The Analogical Mind: Perspectives from Cognitive Science*, pp. 125–159. MIT Press, Cambridge, MA.