Metacat:
A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception

James B. Marshall

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
and the Cognitive Science Program
Indiana University

November 1999

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Dr. Douglas R. Hofstadter
(Principal Adviser)

Dr. Daniel P. Friedman

Dr. David B. Leake

Bloomington, Indiana
September 1999

Dr. Robert F. Port

For the three women in my life:

my mother, my sister, and Francesca.

ABSTRACT

This dissertation describes Metacat, an extension of the Copycat computer model of analogy-making and high-level perception developed by Douglas Hofstadter and Melanie Mitchell as part of a research program aimed at computationally modeling the fundamental mechanisms underlying human cognition. Central to the philosophy of Copycat is the belief that the ability of the human mind to perceive analogies between situations lies at the core of intelligence.

Copycat operates in an idealized microworld of analogy problems involving short strings of letters. The program understands only a limited set of concepts relating to its letter-string world, but its "fluid" conceptual processing mechanisms give it considerable flexibility in recognizing and applying these concepts in many diverse situations.

The present work builds on these achievements by focusing on the issue of *self-watching*—namely, the ability of a system not only to perceive situations, but also to observe and to explicitly characterize its own perceptual processes. Copycat focuses exclusively on perceiving patterns within its input data, while ignoring patterns that occur in its *processing* of those data. Consequently, Copycat lacks insight into how it arrives at its answers. It is thus unable to explain why it considers one analogy to be better or worse than another.

The Metacat project is concerned with extending the model in a way that allows it to create much richer representations of the analogies it makes, enabling it to compare and contrast answers in an insightful way. This involves incorporating an episodic

memory into the architecture, along with an ability for the program to monitor itself, so that it can recognize, remember, and recall important patterns that occur in its own "train of thought" as it makes analogies. By monitoring its own processing, Metacat can recognize when it has fallen into a repetitive pattern of behavior, enabling the program to subsequently break out of the pattern. Furthermore, based on the "meta-level" information gleaned from self-watching, Metacat can come to understand and explain the answers that it finds in a way that Copycat cannot.

ACKNOWLEDGMENTS

What a long and winding road this project has been.

I would like to thank, first of all, my advisor Doug Hofstadter for providing years of generous support and encouragement, as well as constant intellectual inspiration, along the way. It was his book *Gödel, Escher, Bach* that first set me on this path so many years ago, even before I was fully aware of being on it. *GEB* changed my outlook on the world, and had I known, at the time, that I would eventually end up working on the development of a self-watching computer program under the supervision of its author for my PhD, I would have been truly dumbfounded. As it stands, it has been a great pleasure to know him as a friend and mentor throughout my years in graduate school. In addition, I would also like to thank him for making it possible for me to spend a year at the Istituto per la Ricerca Scientifica e Tecnologica (IRST) in Trento, Italy from 1993 to 1994.

The other members of my committee, Dan Friedman, David Leake, and Bob Port, have each had their own unique influence on my intellectual development. Dan has been a true friend from almost the first moment that I arrived in Bloomington, and his exuberant teaching style in C311 taught me much about what it means to be a great teacher. The time I spent as his associate instructor for C311 remains, without a doubt, and for many reasons, one of the most important and rewarding experiences of my graduate school career. I have learned much from David about AI and case-based reasoning. In addition, he was always willing to make himself available whenever I needed to discuss my thesis, or to just vent my frustrations. Bob introduced me to cognitive science through his Q500 intro course back in the early days of my grad school career.

Doug Blank, Gary McGraw, Lisa Meeden, and Jon Rossie have been my closest friends throughout practically my entire time in Bloomington. It is amazing how many times just talking about Metacat with Doug (or drawing illegible diagrams on paper napkins with him over lunch) helped me to figure out what exactly it was that I was trying to do. Gary made life at CRCC fun. Actually, he made life in Bloomington fun. In many ways, the parties that he and Amy had out in their rustic Bean Blossom hideaway remain for me the essence of grad school. Lisa has been such a constant source of intellectual companionship, good humor, and moral support through the years that it is hard for me to imagine what I would have done without her. Being her faculty colleague in the CS program at Swarthmore for the past two years has been truly wonderful. Jon, in addition to being a great friend and erstwhile housemate, has been my link to the programming languages world—the other great interest of mine in CS. Also, hanging out with Jon, Naz, Maddy, and Caleb has provided much-needed fun and relaxation over the years.

Heartfelt thanks go to Melanie Mitchell for writing Copycat in such a remarkably clean and well-organized way that I could actually figure out how it all worked—without having to pester her incessantly with questions (well, at least not too many). I hope Metacat will prove to be a worthy successor to Copycat.

Thanks also to the other FARGonauts, past and present, for making CRCC such a great place to work and hang out (not necessarily in that order), especially John Rehling, Dave Chalmers, Bob French, Liane Gabora, Wang Pei, and Hamid Ekbia.

For invaluable administrative assistance over the years I would like to thank Helga Keller and Pam Larson. They both kept me safely insulated from the IU bureaucracy for years, and for that I am deeply grateful.

Thanks to Jim Herriot for many enjoyable discussions about "Coffeecat" and

"Latte Spirit", and for being such an enthusiastic follower of FARG work in general, and of Metacat in particular. It was always fun to talk about Metacat and Letter Spirit with him and John Rehling whenever he came to town.

Immense thanks go to John Zuckerman for providing CRCC with a copy of his wonderful SchemeXM/SGL graphics package. Without SXM, the development of Metacat would have been infinitely more painful. I am truly more grateful than I can say. Also, John cheerfully made a number of extensions to SXM at my (and Gary McGraw's) request.

Many other friends have enriched my time in graduate school, including Laura Blankenship, Amy Barley, Amy Burton, Manuel Cordero, Patti Freeman, Eric Jeschke, Kannan Konath, Shirley Lee, Mike Ly, Devin McAuley, Brenda Sermersheim, Lisa Thomas, and Paola Voci.

The Swarthmore College computer science department has been an exceptionally congenial place to work for the past two years. Thanks to Charles Kelemen, Lisa Meeden, Jeff Knerr, and Joan McCaul for making my time at Swarthmore so enjoyable, and for waiting patiently while I finished my thesis. In particular, I would like to thank Jeff for letting me borrow an Ultra-5 all summer long (and then some) in order to finish the work. I definitely could not have done it otherwise. Thanks also to the other Swat sysadmins for providing essential software and hardware support.

Thanks to the Pat Metheny Group for providing constant musical inspiration whenever it was needed.

My mother Jean Burris has given me unflagging moral support and loving encouragement throughout my many years of work on this project, and for that I am truly grateful. Thanks, Mom. Likewise, my sister Maria Marshall and my brother-in-law Bill Wightman cheered me on as I approached the finish line, and have been pillars of support throughout the whole process.

Finally, very special thanks go to Francesca Parmeggiani, who has been there with me throughout the entire writing of this dissertation—gently encouraging me to keep going, patiently listening to me complain, and always believing in me no matter what. *Grazie, Chicco (voglio molto bene a te).*

This research has been supported in part by Sun Microsystems Co. Academic Equipment Grant #EDUD-NAFO-960418 and by grants to the Center for Research on Concepts and Cognition from the College of Arts and Sciences of Indiana University.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

The Copycat Model

The Metacat project is an extension of the Copycat computer model of analogy-making and high-level perception originally developed by Douglas Hofstadter and Melanie Mitchell as part of an ongoing research program aimed at computationally modeling the fundamental mechanisms underlying human cognition. Central to the philosophy of this research program is the belief that the ability of the human mind to perceive similarities between different situations—and to make analogies based on these similarities—lies at the core of intelligence. To understand the mental mechanisms by which analogical thinking and perception take place is to understand the source of the remarkable fluidity of the human mind, including its hidden wellsprings of creativity.

1.1 High-Level Perception

Copycat focuses on the idea of *high-level* perception, which can be regarded as that level of perceptual processing in which semantically-defined *concepts* play a critical role [Chalmers et al., 1992]. In contrast, *low-level* perception involves the processing of raw, modality-specific sensory data obtained directly from the environment, such as the processing of light-intensity information from the retina by the visual cortex of the brain, or the processing of auditory information from the inner ear by the temporal cortex. Low-level perceptual processing is a necessary prerequisite for high-level perception to occur, and there are many intermediate processing stages, involving many levels of abstraction, leading from the former to the latter. 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 own 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 successfully recognize one's mother (at least in the absence of degraded environmental conditions such as poor lighting). High-level perception, like many other mental phenomena, depends largely on *subcognitive* processing mechanisms [Hofstadter, 1985c].

The recognition of a person as belonging to the category of *mother* is a fairly prosaic example of high-level perception. This same general phenomenon, however, often occurs at much higher levels of abstraction, such as when a person hears a piece of music for the first time and recognizes it as coming from a particular composer or musical style, or when a painting is clearly recognized to be, say, an Impressionist work, or as belonging to Picasso's "Blue period". Moving to an even higher level of abstraction, an entire situation involving messy and complicated webs of people, objects, relationships, and choices may collectively be seen as a "Catch-22" situation—that is, as an instance of this particular concept. Even the seemingly straightforward concept of *mother* is, in reality, a remarkably subtle matter. Depending on context, a huge variety of things can be recognized as being abstract instances of the *mother* concept. To name just a few examples: the planet Earth is often regarded as being the mother of all living things, an idea commonly expressed by the phrase "Mother Earth". The nation of Russia is often considered to be the mother of the Russian people, as in the phrase “Mother Russia”. Betsy Ross is often described as “the mother of the American flag”. And the Biblical notion of the Apocalypse is sometimes billed as “the mother of all battles”. Normally, one is not inclined to regard planets, nation-states, or military campaigns as likely candidates for motherhood; indeed, in a strict sense such notions do not even make sense. But given the right context, people can effortlessly see how the concept applies, thanks to the natural fluidity of human cognition.

1.2 Conceptual Fluidity and Analogy-Making

In general, a concept in the mind is not a sharply-defined entity with clear-cut boundaries, always applying to certain things (such as someone’s mother) but never to others (such as someone’s father, or a planet). Rather, the boundaries of concepts are inherently ill-defined and blurry, and are strongly influenced by the context in which high-level perception takes place. We refer to this type of inherent flexibility as *conceptual fluidity*, in order to stress the idea of concepts as nonrigid, malleable, adaptable, and highly context-sensitive.

Not only are the “shapes” of individual concepts dynamically adaptable; so too are the “conceptual distances” separating them. To some extent, every concept in the mind consists of a central core idea surrounded by a much larger “halo” of other related concepts. The amount of overlap between different conceptual halos is not rigid and unchangeable, but can instead vary according to the situation at hand. Much work has been done in cognitive psychology investigating the nature of the distances between concepts and categories [Shepard, 1962; Tversky, 1977; Smith and Medin, 1981; Goldstone et al., 1991]. For most people, certain concepts lie relatively close to one another in conceptual space, such as the concepts of *mother* and *father* (or perhaps *mother* and *parent*), while others are farther apart, at least under normal circumstances. However, like the boundaries defining individual concepts, the degree of association between different concepts can change radically under contextual pressure, with the potential result that two or more normally quite dissimilar concepts are brought close together, so that they are both perceived as applying equally well to a particular situation, such as when the Earth is seen as an instance of both the *mother* concept and the *planet* concept. This phenomenon, referred to in the Copycat model as *conceptual slippage*, is what enables apparently unrelated situations to be perceived as being fundamentally "the same" at a deeper, more abstract level.

As another example, consider a typical American adult's concept of *Vietnam*. On the surface, this concept refers to a particular nation located in Southeast Asia. But in the average American mind at least, it is also tightly associated with the huge political and military debacle in which the United States got mired in the late 1960's and early 1970's, along with all of the internal social and cultural strife that occurred as a result. This entire complex political–historical situation is conventionally referred to in America as simply "Vietnam". Hovering about the central core of this concept are enormous numbers of other related concepts, some closer in, some farther out—concepts such as *communism*, *war*, *Cambodia*, *President Nixon*, *social unrest*, *the Pentagon*, *rice*, *1968*, *dominos*, *failure*, and so on. Other concepts clearly lie very far away from the core: *penguins*, say, or *rollerskates*, or *computer software*. Or do they? Consider the following humorous quip, which appeared on the World Wide Web: "Windows 95: Microsoft's Vietnam?" [Leake, 1996]. Whether or not one agrees with its sentiment, its meaning is readily understood. Windows 95, a computer operating system, can be perceived as an instance of the concept of *Vietnam*. In doing so, an implicit analogy has been made between two complex and apparently distinct entities, mapping, among other things, the United States onto Microsoft, the 1960's onto the 1990's, and, perhaps, former U.S. President Richard Nixon onto Microsoft Chairman

Bill Gates. In a more serious vein—and relying on less of a conceptual leap—the 1979 Soviet invasion of Afghanistan is commonly regarded by American political observers as being "the Soviet Union's Vietnam". (Similarly, the 1994–96 conflict in Russia's breakaway Chechnya region might be a strong contender for the title of "Russia's Vietnam".) It is the complex, fluid nature of concepts that allows such analogies to be effortlessly understood and appreciated.

1.3 The Ubiquity of Analogy in Human Thought

Traditionally, researchers working on the computational modeling of analogy have tended to view analogy-making as a special type of thinking useful for solving problems via the technique of *analogical reasoning*. According to this view, a good way to solve a given problem is often by recourse to a similar problem that one has encountered and solved previously. By setting up an analogy between the previous problem and the current problem, and using the previous solution as a guide, one can often discover a solution to the problem at hand [Evans, 1968; Carbonell, 1986; Riesbeck and Schank, 1989; Leake, 1996]. This type of reasoning is often used by students when trying to work through scientific or mathematical problems in textbooks. Typically, a worked-out example in the text, similar to the problem to be solved, is first identified (the more similar, the better). The worked-out example solution is then applied to the corresponding elements of the new problem (hopefully without too much modification required), yielding a solution [Chi et al., 1989].

This type of analogical reasoning is certainly a powerful technique for solving problems, but it is only part of the picture. In contrast to the conscious, deliberative use of analogy as a tool for reasoning about problems, most analogy-making in the mind occurs spontaneously and unconsciously, at the subcognitive level, almost without ever being noticed. To regard analogy-making only as a specialized cognitive tool useful in solving problems is to ignore the ubiquity of analogy in everyday thought processes.

To take just one example from personal experience, I was at a picnic one day, where I had been playing frisbee with several friends. At one point, I approached the serving table where the food was laid out, hungrily eyeing the potato salad, with frisbee in hand, and noticed to my disappointment that there were no more paper plates left. I happened to glance down at the frisbee I was holding, and at that moment I suddenly thought of the idea of using my frisbee as a makeshift plate. In thinking of this, I did not consciously set up a deliberate analogy between the idea of a frisbee and the idea of a plate in hopes of solving the problem of where to put my food; rather, I simply saw the frisbee in a new light, *as a plate*. Another way of saying this is that the particular situation I was in caused me to recognize the object in my hands as an instance of the *plate* concept, whereas under normal circumstances the object causes only the *frisbee* concept to become activated. Said yet another way, a slippage between the concepts of *frisbee* and *plate* occurred in my mind which resulted in the object being perceived simultaneously as an instance of both concepts.

As this example illustrates, the distinctions between categorization, recognition, reminding, and analogy-making are not clear-cut. Rather, all of these mental phenomena represent different types of high-level perception, and are best viewed as points along a broad continuum ranging from simple recognition tasks all the way to highly abstract analogies and poetic metaphors. Robert French gives many wonderful examples of everyday analogy-making running the full length of this spectrum [French, 1995, Chapter 1]. He also discusses the "me-too" phenomenon, an extremely common type of analogy-making that pervades ordinary, day-to-day conversation. Hofstadter has collected a large number of first-hand examples of "me-too" analogies as well [Hofstadter, 1992; Hofstadter and FARG, 1995]. The following exchange between two people having a drink in a hotel lobby—one of whom had a beer and the other a Coke—is typical:

> *Shelley:* I'm going to pay for my beer now.
>
> *Tim:* Me, too.

Such "mundane" analogy-making occurs all the time in human thinking, mostly below the level of conscious awareness. Conscious, step-by-step analogical reasoning in the service of problem-solving is certainly one manifestation of analogy-making, but it is just the tip of the iceberg. Everyday thought and language are permeated with myriad, fleeting analogies effortlessly made and understood, most of which go unnoticed because they seem so unremarkable—such as thinking of a Coke as a beer, or a frisbee as a plate (or describing analogical reasoning as the tip of an iceberg, which is a fairly abstract analogy in and of itself). Indeed, as George Pólya wrote, "analogy pervades all our thinking, our everyday speech and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements" [Pólya, 1957].

1.4 Creativity, Randomness, and Subcognition

The connection between analogy-making and scientific creativity has long been recognized. Analogies have played an instrumental role in the creation of new and sometimes revolutionary scientific theories. One of the most famous examples, discussed at length by Margaret Boden in her book *The Creative Mind: Myths and Mechanisms* [Boden, 1991], was the discovery of the molecular structure of the benzene molecule by Friedrich von Kekulé in 1865, who, while dozing by the fireside, experienced a vision of a snake biting its own tail. In a flash, he realized that a molecular *ring* structure was the answer to the mystery of the benzene molecule's geometry, which, up until that time, had been assumed by chemists to consist of a linear sequence of atoms. This insight led to the establishment of the field of aromatic chemistry. As this example shows, the capacity for creative, insightful thinking is deeply tied to the capacity for perceiving abstract similarity between things that, on a more concrete level, would appear to be utterly different. Keith Holyoak and Paul Thagard's book *Mental Leaps: Analogy in Creative Thought* contains a good discussion of many other instances of analogically-inspired creativity in science [Holyoak and Thagard, 1995, Chapter 8].

In her enlightening book, Boden examines many historical examples of creativity taken from art, music, science, and literature. In almost all verbal or written accounts of the creative process, whether scientific, artistic, or otherwise, the actual moment of insight experienced by the creator seems to take on an almost mystical or indescribable aura. It seems notoriously difficult to pin down, in any precise way, exactly which thought processes are involved in the creative act itself. Indeed, it often seems as if new ideas come randomly, without warning, from "out of the blue". As the mathematician Jacques Hadamard put it [Koestler, 1964]:

> On being very abruptly awakened by an external noise, a solution long searched for appeared to me at once without the slightest instant of reflection on my part.

Both Henri Poincaré and Arthur Koestler were interested in the underlying mental mechanisms of creativity that give rise to such seemingly unanalyzable flashes of insight. Poincaré expressed a view of creativity as the random coming together of diverse ideas in the subconscious mind, much like a swarm of gnats or a collection of gas molecules jostling against one another [Poincaré, 1921]. Koestler, however, maintained that a purely random association of ideas is not enough; the mixing of ideas must be guided by mental structures acting as constraints, a process he termed the "bisociation of conceptual matrices" [Koestler, 1964]. Indeed, this notion of directed randomness in the service of creativity turns out to be of critical importance in understanding the fluid mechanisms of cognition, and consequently plays a central role in the architecture of the Copycat program.

On the other hand, both Koestler and Poincaré agreed that the processes involved in creative thinking are carried out largely at the subconscious level. As Koestler eloquently observed [Koestler, 1964]:

> The moment of truth, the sudden emergence of a new insight, is an act of intuition. Such intuitions give the appearance of miraculous flashes, or short-circuits of reasoning. In fact they may be likened to an immersed chain, of which only the beginning and the end are visible above the surface of consciousness. The diver vanishes at one end of the chain and comes up at the other end, guided by invisible links.

In fact, Poincaré distinguished four phases of creativity that occur during problem-solving: an initial preparatory phase involving conscious attempts to solve the problem using familiar methods; an incubation period in which the conscious mind is focused on other things while at the same time ideas are being continually recombined at a deeper, subconscious level; an abrupt flash of insight at the level of conscious awareness; and, finally, an evaluation phase in which the insight's ramifications are consciously worked out in full. He characterized the sudden flash of illumination as "a manifest sign of long, unconscious prior work".

1.5 A Computer Model of Conceptual Fluidity

A fundamental motivation driving the Copycat project, and consequently the Metacat project, is a belief in a common set of mechanisms responsible for creative insights and analogy-making—from the most rarefied strokes of artistic and scientific genius all the way down to the (far more common) type of run-of-the-mill analogy-making that pervades everyday thought and language. According to this view, creative analogical thought is a natural by-product of the dynamic, fluid nature of concepts in the mind.

Conceptual fluidity provides the means through which flexible high-level perception takes place.

Copycat is a computer model of the nondeterministic, subcognitive mental processes out of which conceptual fluidity emerges. It represents the tangible instantiation of a general theory of mind describing how fluid concepts interact with and guide perception, and how genuine understanding—at least in a limited domain—can emerge from the dynamics of this interaction. A fully detailed exposition of the Copycat project can be found in [Mitchell, 1993] and [Hofstadter and FARG, 1995]. The rest of this chapter gives a thorough, but condensed, overview of Copycat, along with several examples illustrating the behavior of the program. Since the Metacat project builds directly on the Copycat architecture, this background is necessary in order to understand the work on Metacat described in the remainder of this dissertation.

1.5.1 An idealized microworld for studying analogy-making

The domain in which Copycat operates is a *microdomain*—a tiny, idealized world explicitly designed to isolate the essential, fundamental aspects of analogy-making and creativity by stripping away from it as many insignificant and confusing "real-world" details as possible. This act of idealization brings out the deep issues of high-level perception in stark relief, rendering them accessible and amenable to careful, controlled study.

Specifically, the raw material of Copycat's domain consists of the 26 lowercase letters of the alphabet. Copycat analogy problems are stated in terms of three strings of letters: the *initial string*, the *modified string*, and the *target string*. A typical Copycat problem is the following: "If the string ***abc*** changes to the string ***abd***, how might the string ***mrrjjj*** change in an analogous way?" Or, displayed graphically:

abc $\Rightarrow$ ***abd***
mrrjjj $\Rightarrow$ ***?***

On first seeing this problem, most people answer either ***mrrkkk*** or ***mrrjjk*** [Mitchell, 1993]. The rightmost component of ***abc*** (*i.e.*, 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 (*i.e.*, either the group of three ***j***'s viewed as a chunk, or just the rightmost letter ***j***).

There are, however, many other defensible answers to this problem, which people tend to give less often, including:

- ***mrrjjd*** (change the rightmost letter literally to ***d***)
- ***mrrddd*** (change the entire rightmost group to ***d***'s)
- ***mrrjjj*** (change only ***c***'s to their successor)
- ***mrrjkk*** (view ***mrrjjj*** as the three letter-pairs ***mr–rj–jj*** and change the rightmost pair to its successor)
- ***mrrjdd*** (view ***mrrjjj*** 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***)
- ***mrrjjjj*** (view ***mrrjjj*** abstractly in terms of group lengths, as *1–2–3*, and increase the rightmost length by one)
- ***mrrkkkk*** (view ***mrrjjj*** as *1–2–3*, but change both the length and letters of the rightmost group)
- ***mrsjjk*** (view ***mrrjjj*** as ***mrr–jjj*** and change the third letter of each group to its successor)

- ***mrskkk*** (change each letter after the two leftmost letters to its successor)
- ***mssjjj*** (change each instance of the third letter to its successor)
- ***abd*** (change the entire 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*** to ***k*** but make everything single letters)
- ***mrd*** (change ***j*** to ***d*** but make everything single letters)

Clearly, some of these answers are more obvious than others, and the obvious ones may not be the most aesthetically pleasing ones, but there is no single, indisputably "correct" answer. In fact, a wide range of answers is possible for almost any imaginable Copycat problem. Despite its apparent simplicity, Copycat's domain harbors an exceedingly rich variety of subtle analogy problems often admitting deeply elegant yet non-obvious answers [Hofstadter, 1984b; Hofstadter, 1985a; Mitchell, 1993]. This is the mark of a well-designed microdomain. The depth and complexity of analogy-making in the Copycat world has not been sacrificed at the expense of simplicity; on the contrary, the deep issues of analogy-making have been brought to the surface and laid bare, precisely because of the domain's austerity.

An analogy from physics may be useful in thinking about the utility of microdomains. In order to understand complex physical phenomena occurring in the real world, physicists first devise idealized theoretical models of the phenomena, in an attempt to understand their most essential aspects. Only after these fundamental aspects have been understood can further progress be made in understanding the full complexity of the phenomena as they occur in the real world. For instance, in order to understand the complex motion of real-world objects, it is necessary to first understand the motion of objects in an idealized, frictionless world. Ignoring the complicating factor of friction allows the fundamental laws of motion to be understood, which in turn provides the foundation necessary for achieving a deeper understanding of motion involving friction.

Likewise, modeling human analogy-making or other "real-world" cognitive phenomena in a microdomain necessarily involves selectively ignoring certain aspects of cognition, while concentrating on others of more fundamental importance. This does not mean, however, that the former aspects are unimportant or unworthy of investigation—only that they are better left for later investigations, after a deep understanding of the truly fundamental aspects of cognition has been achieved.

In the "frictionless" world of Copycat's microdomain, the fundamental aspects under study are the fluid nature of concepts and the phenomenon of conceptual slippage. Other aspects of cognition have been deliberately idealized away, such as the retrieval of knowledge from a large repertoire of experience stored in memory, or the learning of new concepts from experience. To be sure, no full and satisfying account of cognition will be possible without a deep understanding of these latter phenomena. However, such an account—whenever it may come—will surely rest on a deep understanding of the nature of concepts in the mind.

Like the idealized models of physics, Copycat's microdomain facilitates the study of concepts and analogy-making by avoiding many types of "friction" that would otherwise further compound the difficulty of understanding these mental phenomena. In the meantime, it remains a formidable challenge, indeed, to develop a computer program capable of displaying the full range of creativity and flexibility exhibited by people on problems taken from this domain—however tiny and idealized it may be.

It is also important to stress the generality of Copycat's domain. The letter-string microworld has been carefully designed with an eye toward *universality.* All information pertaining specifically to letters has been factored out, such as the actual shapes of letters or any associated semantic connotations. In the Copycat world, letters are nothing more than abstract, atomic categories, much like the notion of an undefined term in geometry. It is irrelevant whether or not letter-strings happen to form recognizable words or phrases. Furthermore, only three relations among letters are meaningful: sameness, predecessorship, and successorship. All letters except ***a*** have an immediate predecessor, and all letters except ***z*** have an immediate successor; hence the alphabet does not "wrap around" from ***z*** back to ***a***.[1] No other properties of letters are involved.

Thus, it is misleading to regard Copycat analogy problems as being about alphabetical strings of letters *per se*. Rather, they should be viewed simply as idealized situations involving a set of abstract objects, among which certain relations may hold. The architecture of Copycat is "configured" so that these objects and relations correspond to our intuitive notions about successorship, predecessorship, and sameness among letters of the alphabet, but this need not be the case. A different configuration could be chosen to reflect a different set of objects and relationships, without significantly altering the architecture of the program. In fact, a program similar to Copycat, called Tabletop, models certain spatial aspects of high-level perception that occur in a different domain: that of objects on an ordinary table, such as cups, glasses, and silverware [Hofstadter and French, 1992; French, 1995; Hofstadter and 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 domain of objects and relations. Copycat is configured so that these objects and relations mirror certain properties of letters of the alphabet, while Tabletop is configured so that they mirror certain properties of objects on a table.

Copycat's microdomain is sometimes criticized as being unable to represent analogies between different domains of knowledge. So-called "cross-domain" analogies—for

[1] The choice of a strictly linear alphabet was made deliberately, in order to introduce structural irregularity to the domain. This gives an interesting twist to certain analogy problems involving the "edge" letters ***a*** and ***z***, to be discussed later.

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 and Thagard, 1989; Falkenhainer et al., 1990]—typically involve source and target situations characterized by very different subsets of "real-world" concepts. The true power of analogy manifests itself in such mappings through the transfer of useful ideas between apparently dissimilar domains. In contrast, it is argued, since Copycat's source and target situations are both restricted to letter-string concepts only, the model is "domain-specific", and hence fails to capture the most important aspects of analogical processing. As [Forbus et al., 1998] puts it:

> 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. (page 247)
>
> ...
>
> 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. (page 251)

However, such a hasty conclusion overlooks the principle of universality at the core of Copycat's letter-string domain. Since the "letters" of Copycat's world—as far as the program is concerned—are really just atomic categories joined together by abstract relationships, there is in principle no reason why idealized versions of "cross-domain" analogies cannot be constructed within this microdomain as well.

Indeed, the answer ***mrrjjjj*** to the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"* mentioned earlier (which will be discussed more fully in section 1.5.4) could be interpreted as just such an analogy. On the surface, different sets of ideas apply to the situations represented by the strings ***abc*** and ***mrrjjj***. For example, the idea of *successorship* is clearly present in the former situation, while the notion of a *group* is central to the latter. In an abstract sense, these strings could be viewed as situations taken from two very different domains, each of which involves a subset of the concepts available in the encompassing letter-string microdomain. If the two situations are looked at in the right way, however, 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 hidden *1–2–3* structure of ***mrrjjj***, which consequently leads to the answer ***mrrjjjj***. Of course, both of these "domains" involve concepts taken from Copycat's letter-string world, but the crucial point is that they involve *different* sets of concepts, just as the domains of "cross-domain" analogies from the real world involve different subsets of real-world concepts taken from the larger universe of real-world things and relationships.

In fact, on closer examination, the distinction between different domains is often not clear-cut. For instance, Holyoak and Thagard discuss a complex analogy between World War II and the Persian Gulf War [Holyoak and Thagard, 1995]. 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 models?[2] The purported distinction between "cross-domain" and "intra-domain" analogies (as well as the distinction between "domain-general" and "domain-specific" approaches to modeling analogy) is in fact largely artificial, and depends very much on the particular definition of the

[2] The performance of the Structure-Mapping Engine (SME) program, developed by Brian Falkenhainer, Ken Forbus, and Dedre Gentner, on this particular analogy problem has often been cited as support for the claim that SME can handle analogies between very different real-world domains [Falkenhainer et al., 1990]. See [Hofstadter and FARG, 1995] for a detailed examination and discussion of these claims, as well as similar claims made by Holyoak and Thagard about their Analogical Constraint Mapping Engine (ACME) program—another model of analogy supposedly able to make cross-domain mappings [Holyoak and Thagard, 1989].

domains involved, which in turn depends on the particular way in which we choose to carve up the world into categories. The power of a microdomain derives from its ability *in principle* to model any number of different aspects or domains of the real world within a common abstract framework.

1.5.2 The architecture of Copycat

The Copycat architecture is divided into two principal components, which can be thought of as corresponding very roughly to short-term and long-term memory. Copycat's "short-term memory", called the *Workspace*, serves as the locus of perceptual activity during a run of the program. However, in contrast to human short-term memory, information in Copycat's Workspace cannot be transferred to "long-term memory" or otherwise retained indefinitely. The information in the Workspace is specific to each individual run, and has no effect on subsequent runs of the program.[3] The Workspace is similar to the Blackboard architectural component of the Hearsay II speech-understanding system, from which the Copycat project derived much early inspiration [Erman et al., 1980].

When Copycat is given an analogy problem to work on, it starts out with the letter-strings in its Workspace. Small, nondeterministic computational agents called *codelets* notice relations among the individual letters and build new structures around them, effectively organizing the letters into a coherent high-level picture. Codelets "swarm" about the Workspace looking for suitable structures to work on, much like enzymes in a cell. All processing occurs through the collective actions of many codelets working in parallel, at different speeds, on different aspects of an analogy problem, without any centralized "executive" controlling the course of events. The overall macroscopic

[3]Addressing this deficiency is a central goal of the Metacat project, and will be discussed further in Chapter 2.

behavior of the program is not explicitly programmed; rather, is a statistically emergent consequence of a large number of stochastic, localized micro-actions performed by codelets.

Influencing the perceptual activity occurring in the Workspace are *active concepts*, which reside in the "long-term memory" component of Copycat, called the *Slipnet* (shown schematically in Figure 1.1). Most perceptual structures in the Workspace are, in fact, instances of particular concepts in the Slipnet (such as *letter* or *group*). The Slipnet serves as the program's permanent repository of knowledge about its domain. It contains representations for various concepts relevant to solving letter-string analogies—such as *successor* and *predecessor*, the abstract notion of *opposite*, the letter-categories *a*, *b*, *c*, and so on—as well as a numerical estimate of the intrinsic degree of abstractness of each concept, called the concept's *conceptual depth* (not shown in the figure). The Slipnet also encodes information about the inherent associative distances between concepts, which determine the propensities for various conceptual slippages to occur. A slippage between a pair of Slipnet concepts occurs whenever instances of the concepts in the Workspace are seen as playing identical roles in different contexts.

Some concepts in the Slipnet are themselves instances of other concepts. For example, the concepts *left* and *right* are both instances of the more abstract *Direction* concept, and the concepts *leftmost*, *rightmost*, *middle*, *single*, and *whole* are all instances of the *String-Position* concept. Nodes that represent various categories of concepts, such as *Direction* or *String-Position*, are called *category nodes*, and are shown capitalized in the figure.

Although the Slipnet contains permanent information, it is not a static structure. Over the course of a run, concepts in the Slipnet assume different levels of *activation*; as this happens, distances between concepts grow and shrink, changing the propensities for various slippages to occur. The stochastic behavior of codelets is dynamically

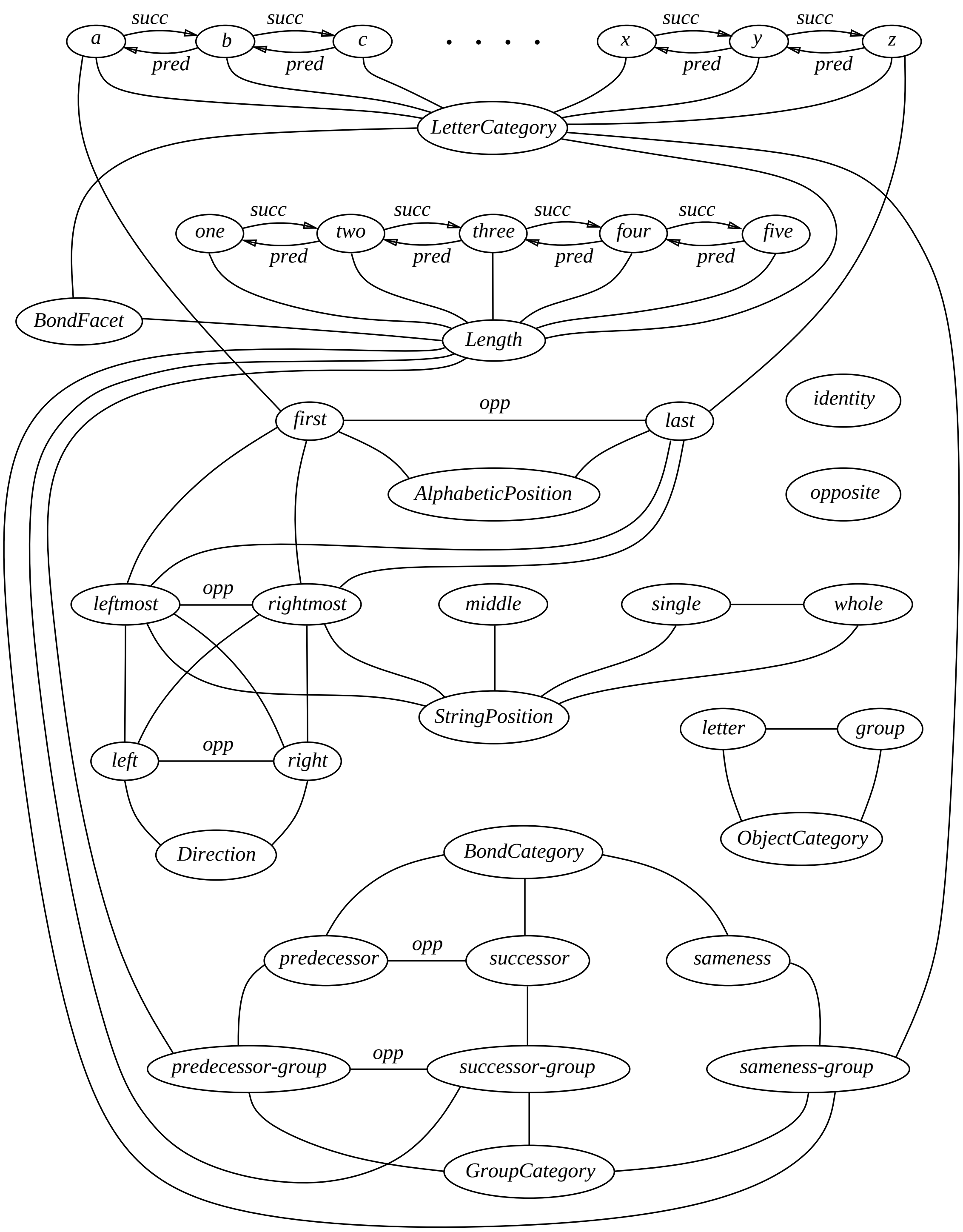


Figure 1.1: *Copycat's Slipnet*

biased by the time-varying pattern of concept activations in the Slipnet. In turn, this pattern of activations is itself an emergent consequence of codelet processing. Conceptual activity in the Slipnet thus influences, and is influenced by, perceptual activity in the Workspace, forming a tightly-coupled feedback loop between these two architectural components.

1.5.3 Conceptual activity in the Slipnet

In some ways, the Slipnet is similar to a traditional semantic network, in that it consists of a set of nodes connected by links. Each of these links has an intrinsic *length* that represents the general degree of association between the linked nodes, with shorter links connecting more strongly associated nodes (the links drawn between nodes in Figure 1.1 do not in general reflect the actual lengths involved). Each node corresponds to an individual concept, or rather, to the *core* of an individual concept. A concept is more properly thought of as being represented by a diffuse region in the Slipnet centered on a single node. Nodes connected to the core node by links are included in the central node's "conceptual halo" as a probabilistic function of the link lengths. This allows single nodes to be shared among several different concepts at once, depending on the links involved. Thus, concepts in the Slipnet are not sharply defined; rather, they are inherently blurry, and can overlap to varying degrees.

Unlike traditional semantic networks, however, the Slipnet is a dynamic structure. Nodes in the Slipnet receive frequent infusions of activation, as a function of the type of perceptual activity occurring in the Workspace. Activation spreads throughout a node's conceptual halo, flowing across the links emanating from the core node to its neighbors. The amount of spreading activation is mediated by the link lengths, so that more distant nodes receive less activation. However, the link lengths themselves are not necessarily fixed. Some links are *labeled* by particular Slipnet nodes, and may stretch or shrink in accordance with the activation of the label node. A labeled link encodes a specific type of relationship between two concepts, in addition to the conceptual distance separating them. For example, the link between the *predecessor* and *successor* nodes is labeled by the *opposite* node, and the link from the *a* node to the *b* node is labeled by the *successor* node. Whenever a node becomes strongly activated, all links labeled by it shrink. As a result, pairs of concepts connected by these links are brought closer together in the Slipnet, allowing activation to spread more easily between the two, and also making it more likely for conceptual slippages to occur between them.

In the absence of further infusions of activation, a node's activation level gradually decays towards zero at a rate that depends inversely on its conceptual depth. Thus, shallow, "surface-level" concepts such as *a* tend to decay more rapidly than highly abstract concepts like *opposite*. As a node's activation decays, any links labeled by it relax back to their intrinsic lengths. The Slipnet thus has a decidedly "spongy" feel to it, reflecting the fluid nature of the concepts it represents. Slipnet concepts "awakened" by perceptual activity occurring in the Workspace, and to a lesser degree neighboring concepts awakened through spreading activation, distort the overall "shape" of the Slipnet, temporarily blending and blurring various concepts into one another. Driven by the gradual ebb and flow of activation, new patterns of active concepts continually emerge in the Slipnet, deforming and reshaping it anew, throughout the course of a program run.

1.5.4 Perceptual activity in the Workspace

Conceptual activity in the Slipnet influences the behavior of codelets as they build new structures in the Workspace. These structures include *bonds* representing successor, predecessor, or sameness relations between adjacent letters of a string; *groups* composed of adjacent letters (or possibly other groups) that have been bonded together by a common relation; *bridges* between letters or groups in different strings;

various types of *descriptions* of structures; and a *rule* describing the way in which the initial string changes into the modified string.

Bonds and groups bind the individual letters of a string (*i.e.*, the raw, unstructured perceptual input) together into hierarchical chunks. For example, during a typical run of the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ ?"*, sameness bonds are created between the adjacent ***j***'s and the pair of ***r***'s in ***mrrjjj***. These bonds then serve as the basis for creating two *sameness groups*, ***rr*** and ***jjj***.

Codelets, in addition to building up the internal organization of strings by chunking letters into groups, also build *mappings* between strings. A mapping consists of a set of bridges between letters or groups in two different strings that play similar roles in each string. Each bridge is supported by a set of *concept-mappings* that describe how the objects connected by the bridge correspond to one another. For example, a bridge between ***c*** in ***abc*** and ***jjj*** in ***mrrjjj*** might be supported by the concept-mapping *letter* ⇒ *group* (a slippage representing the idea that one object is a letter and the other a group), and the concept-mapping *rightmost* ⇒ *rightmost* (an "identity mapping" representing the idea that both objects are rightmost in their strings). The distributed nature of codelet processing interleaves the mapping process with the chunking process, and as a result each process influences and drives the other.

Codelets also build rules, which are Workspace structures representing how the initial string changes into the modified string.[4] There are usually several possible ways of describing this change, depending on the level of abstraction used. For example, two possible rules describing ***abc*** ⇒ ***abd*** are *Replace letter-category of rightmost letter by successor* and the more "literal-minded" rule *Replace letter-category of rightmost letter by 'd'*.

Whenever new Workspace structures are built, concepts in the Slipnet relating to

[4]This usage of the term "rule" differs significantly from the traditional AI meaning of the term. In particular, Copycat's rules are completely unrelated to the types of rules used in expert systems or other rule-based problem-solving systems.

them receive activation, which then spreads to neighboring concepts. In turn, highly-activated concepts exert *top-down pressure* on subsequent perceptual processing by promoting the creation of new instances of these concepts in the Workspace. Thus, which types of new Workspace structures get built depends strongly on which concepts are relevant (*i.e.*, highly activated) in a given context.

For example, the creation of the groups ***rr*** and ***jjj*** in ***mrrjjj*** causes the *sameness* and *sameness-group* concepts in the Slipnet to become highly activated, which makes it more likely that the letter ***m*** itself will be seen as a sameness group as well, even though it consists of just a single letter. Given the context of the ***rr*** and ***jjj*** sameness groups, seeing ***m*** as a group of length one makes sense. The creation of such a group causes the *Length* concept—up until now deemed irrelevant to the situation—to become activated in the Slipnet. Once the relevance of this idea has been recognized, a higher-level *successor group* composed of ***m***, ***rr***, and ***jjj*** encompassing the entire string can then be built, based on the concept of *Length* (*i.e.*, *1–2–3*) rather than *Letter-Category* (*i.e.*, *m–r–j*). This brings out the abstract successorship structure of ***mrrjjj***, allowing it to be mapped as a whole onto the letter-category-based successor group ***abc***, which leads to the answer ***mrrjjjj***. Such a mapping represents the recognition of ***abc*** and ***mrrjjj*** as being fundamentally the same at a deep level, even though their surface resemblance is negligible. Figure 1.2 shows the final activations of concepts in Copycat's Slipnet at the end of a run in which the program found the answer ***mrrjjjj***.[5] The size of a circle represents the activation level of a node. In particular, the *successor-group* node is highly activated, reflecting the relevance of this concept in the current context.

[5]Copycat was originally implemented in Common Lisp and C for the SunView window system. The version of the program shown here is a complete reimplementation written in Scheme for the X window system using John Zuckerman's superb SchemeXM/SGL package, an extended symbolic graphics language for X/Motif based on Chez Scheme [Zuckerman, 1992a; Zuckerman, 1992b; Dybvig, 1996].

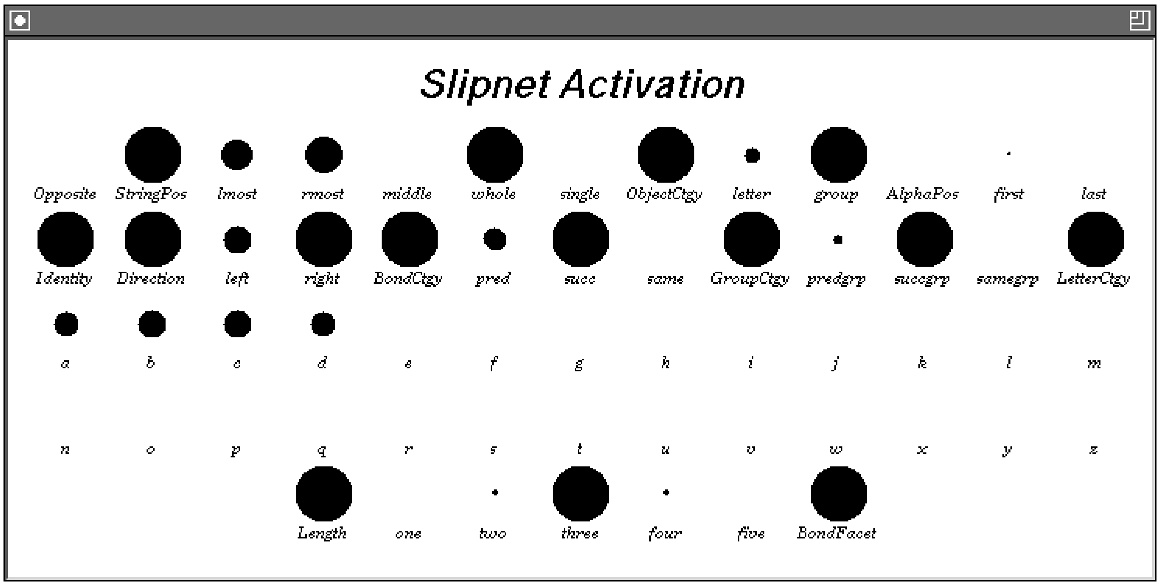


Figure 1.2: *The final activations of Slipnet concepts for a run of Copycat on the problem "****abc****⇒* ***abd****;* ***mrrjjj****⇒ ?", in which the program found the answer* ***mrrjjjj****.*

Different ways of looking at the initial/modified change, combined with different ways of building the initial/target mapping, give rise to different answers. The configuration of structures in the Workspace collectively represents an *interpretation* of a given analogy problem, and leads to a particular answer for the problem. To produce an answer, codelets use the slippages underlying the initial/target mapping to "translate" the rule describing the initial/modified change into a new rule that applies to the target string.

For example, if the ***abc*** ⇒ ***abd*** change is described as *Replace letter-category of rightmost letter by successor*, and the abstract successor-group similarity between ***abc*** and ***mrrjjj*** has been noticed, then the rule will be translated as *Replace length of rightmost group by successor*, yielding the answer ***mrrjjjj***. On the other hand, if this similarity has not been noticed—that is, if the mapping between ***abc*** and ***mrrjjj*** does not include a bridge supported by the slippage *Letter-category* ⇒ *Length*—then

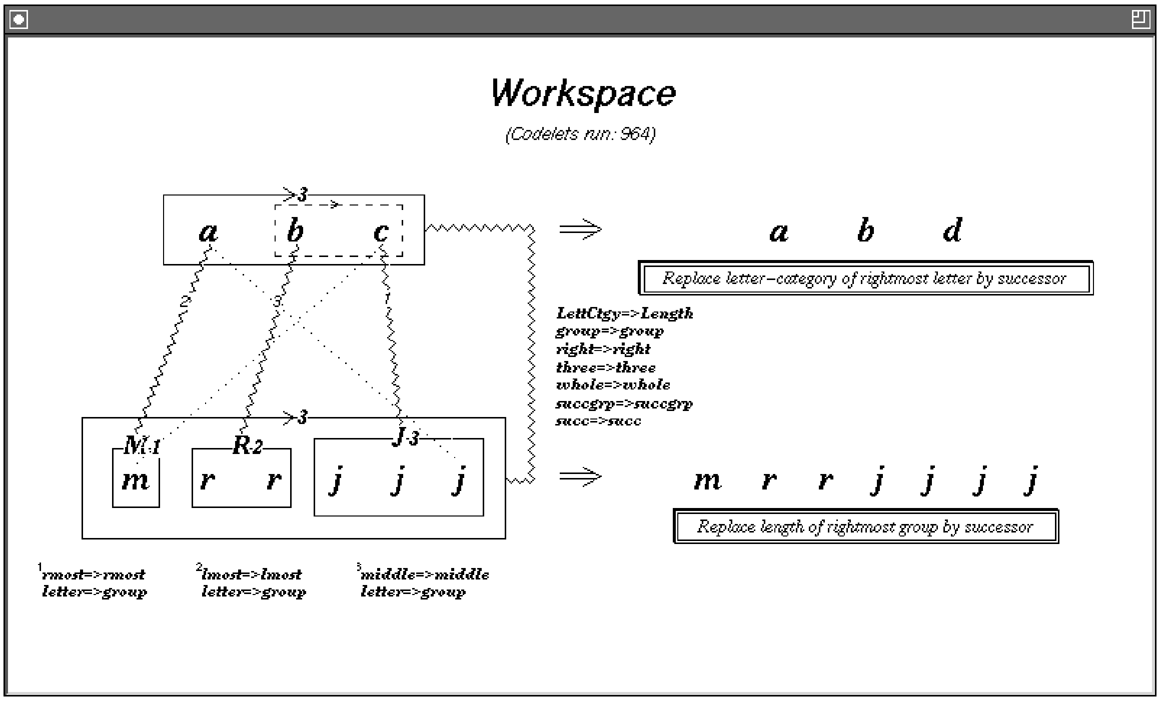


Figure 1.3: *The final Workspace configuration for a run of Copycat on the problem "**abc** ⇒ **abd**; **mrrjjj** ⇒ ?", in which the program found the answer **mrrjjjj**.*

other answers such as ***mrrkkk***, ***mrrjjk***, ***mrrddd***, or ***mrrkkd*** may be found instead, depending on the rule and whether or not ***c*** in ***abc*** is seen as corresponding to the ***jjj*** group or to just the rightmost letter ***j*** in ***mrrjjj***. Figure 1.3 shows the Workspace at the end of a run in which ***mrrjjjj*** was found. (This is the same run that was referred to in Figure 1.2.) Many structures can be seen, including concept-mappings supporting the vertical bridges between ***abc*** and ***mrrjjj***, the rule describing ***abc*** ⇒ ***abd***, the translated rule describing ***mrrjjj*** ⇒ ***mrrjjjj***, and several tentative structures that were being explored but had not yet been built by codelets (shown as dotted lines). A complete discussion of Copycat's behavior on this problem, including screen dumps of sample runs, can be found in [Mitchell, 1993] and [Hofstadter and FARG, 1995].

1.5.5 Codelets and the parallel terraced scan

In general, the letter-strings of an analogy problem can be interpreted in many different, often mutually exclusive ways. For most problems, the potential number of distinct configurations of bonds, groups, bridges, and so on, is very large. If Copycat were simply to try out every possible configuration, one after the other, trying to find a compelling interpretation of the strings, it would be quickly overwhelmed by a combinatorial explosion of possibilities. Instead, in order to discover a good overall configuration of structures from among a vast set of possibilities within a reasonable amount of time, many potential pathways through "interpretation space" must be searched simultaneously, with relatively more attention being devoted to exploring promising pathways than to pathways 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 underlying the Copycat architecture.

To achieve this differential effect, Workspace structures are built in stages rather than all at once. At first, a structure is simply proposed as a possible candidate by codelets. This tentative structure subsequently undergoes an evaluation stage, in which its potential for strengthening the existing perceptual organization in the Workspace is estimated. Finally, if the structure seems promising enough, it gets built, and acquires a *strength* value indicating how well it fits into its surrounding context. The presence of the newly-built structure may in turn alter the strengths of other structures in the Workspace, or the activation levels of concepts in the Slipnet, thereby changing the perceptual context and consequently influencing the fate of other tentative structures still in the early stages of creation.

Since any structure must pass through several stages during its creation, all structures are ultimately built by *chains* of codelets, rather than by single codelets. Codelets responsible for proposing new structures or evaluating proposed structures spawn new codelets, which then continue the process at the next stage in the chain.

Distributing the process of structure creation over several stages is critical, because the interleaving of these stages allows many mutually-dependent processes to effectively run in parallel, exploring the search space in various directions simultaneously. Perceptual activity in the Workspace consists of a large number of these tightly intertwined, concurrent exploratory processes.

Sometimes a structure is proposed that would be incompatible, if built, with an existing structure. In "***abc*** ⇒ ***abd***; ***mrrjjj*** ⇒ ***?***", for instance, the ***c*** cannot correspond to both the ***jjj*** group and the single rightmost letter ***j*** at the same time, since this would make no sense in an analogy. Bridges representing these correspondences are mutually incompatible. An existing structure in the Workspace may, in fact, be destroyed in favor of a new, more promising one if the existing structure has a low strength value relative to the proposed structure.

In addition to its strength, each structure has a *salience* value that determines how much it tends to attract attention from codelets. More specifically, codelets choose Workspace structures for processing as a probabilistic function of their saliences.

Some types of codelets are concerned with general *bottom-up* properties of the input data, while other types are driven by specific *top-down* contextual pressures.[6] For instance, some bottom-up codelets examine adjacent letters or groups to see if *any* type of bond can be made between them, regardless of the current perceptual context. In contrast, top-down codelets look for ways to build structures that support a particular concept. If, for example, several sameness groups have been built in a string (as in ***mrrjjj***, described earlier), the sameness concept will be strongly activated in the Slipnet. In this context, there is greater pressure to notice sameness relations, if they exist, than successor or predecessor relations. The active sameness concept floods the Workspace with top-down codelets specifically looking for sameness

[6] All in all, there are 24 different types of codelets in Copycat. See [Mitchell, 1993] for a detailed description of each type.

among letters, increasing the likelihood that other sameness bonds will be created. In general, top-down codelets driven by context-sensitive pressures are the means through which conceptual activity in the Slipnet influences perceptual activity in the Workspace.

Because Copycat is implemented on a serial computer, codelets have to be run one at a time. In order to realize the differential parallelism of the parallel terraced scan, a pool of available codelets is maintained, called the *Coderack*, from which codelets are selected probabilistically to run. Each codelet in the Coderack is assigned an *urgency* value reflecting the codelet's estimated promise of the pathway it is exploring. Codelets are selected to run as a stochastic function of their urgencies, and as a result, promising regions of Copycat's search space tend to be explored more quickly and to a greater depth, on average, than less promising regions, although even the lowest-urgency codelets always have some chance of running. This is important, because in principle all regions of the search space should always remain open to the possibility of exploration, even if they do not currently appear to be interesting. This type of urgency-modulated stochasticity, which allows different processes to advance at different rates according to their estimated promise, gives rise to the parallel terraced scan.

1.5.6 Temperature and nondeterminism

The nondeterministic nature of Copycat's processing implies that different runs of the program on the same analogy problem may produce different answers. Indeed, the program is usually able to discover a range of answers for any given problem. If Copycat is run many times on a single problem, clear trends emerge. Typically, the program finds one or two answers much more frequently than it finds other answers. These answers are, in some sense, more "obvious" to the program, and lie in easily accessible regions of Copycat's search space. For example, Figure 1.4 shows

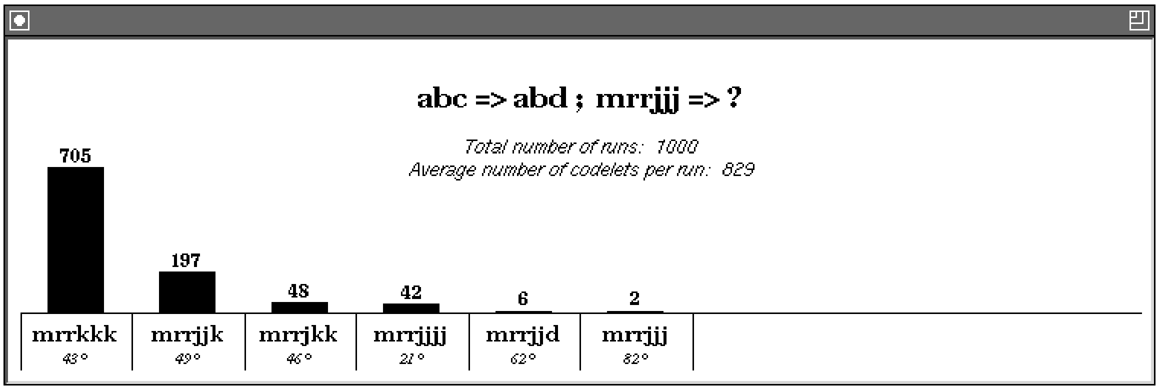


Figure 1.4: *Summary of 1000 runs of Copycat on the problem "**abc**⇒ **abd; mrrjjj**⇒ **?**", showing each answer's frequency and average final temperature. From [Mitchell, 1993].*

a histogram of Copycat's answers for the problem *"**abc**⇒ **abd; mrrjjj**⇒ **?**"*. The straightforward answer ***mrrkkk*** is by far the most common—both for Copycat and for people [Mitchell, 1993].

In general, the most obvious answer is not necessarily the "best" answer. The notion of answer quality is represented in Copycat by a dynamically changing number called the *temperature* (ranging from 0 to 100), which reflects the degree of perceptual order in the Workspace. At the beginning of a run, when few perceptual structures exist, the temperature of the Workspace is very high, reflecting a general absence of understanding of the input strings. Gradually, as codelets examine the situation and build new structures, increasing the perceptual organization of the Workspace, the temperature falls, reflecting a more coherent understanding of the strings. If, however, structures are subsequently destroyed, the temperature will increase. At the end of a run, the final Workspace temperature can be interpreted as a measure of the quality of the answer found, with lower temperatures indicating higher quality. An insightful answer—one based on a strong, coherent mapping between the initial and target strings—typically has a very low final temperature. For example, the average final temperatures of Copycat's answers for the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"* appear immediately below each answer in Figure 1.4.

Temperature does more than simply reflect the ever-changing degree of order in the Workspace. It also continually *influences* the many probabilistic decisions made by codelets throughout the course of processing. Temperature acts as a focusing mechanism for the search process by dynamically regulating the amount of randomness used in making decisions. At high temperatures, it is hard to distinguish promising from unpromising directions, since little structural information exists in the Workspace. As a consequence, decisions are made in a highly random manner, with codelet urgencies, as well as the strengths and saliences of existing structures, having only a marginal effect. However, as regularities among the letter-strings are discovered and structures are built, Copycat begins to gain "confidence" in its understanding of the situation, and less randomness seems called for in making decisions. At lower temperatures, therefore, decisions are still stochastic, but are more strongly biased according to current urgencies, saliences, and strengths. At very low temperatures, decisions become largely deterministic, with the highest-urgency codelets almost always being chosen to run next, the most salient structures almost always being looked at, and so on. Thus, the type of strategy Copycat uses to explore its search space ranges along a broad continuum, from being very diffuse and highly parallel at high temperatures to being very serial and focused at low temperatures.

As an illustration of this, Figure 1.5 shows the state of Copycat's Coderack at two different points during the earlier run of the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"* from Figure 1.3. For each possible codelet type, the relative probability that a codelet of that type will be selected to run next is indicated by a horizontal bar. The left image shows the selection probabilities of codelets at an early point in the run, when the temperature is high. As can be seen, the selection probabilities are all roughly the same, reflecting the still-strongly-parallel nature of processing. In contrast, the right

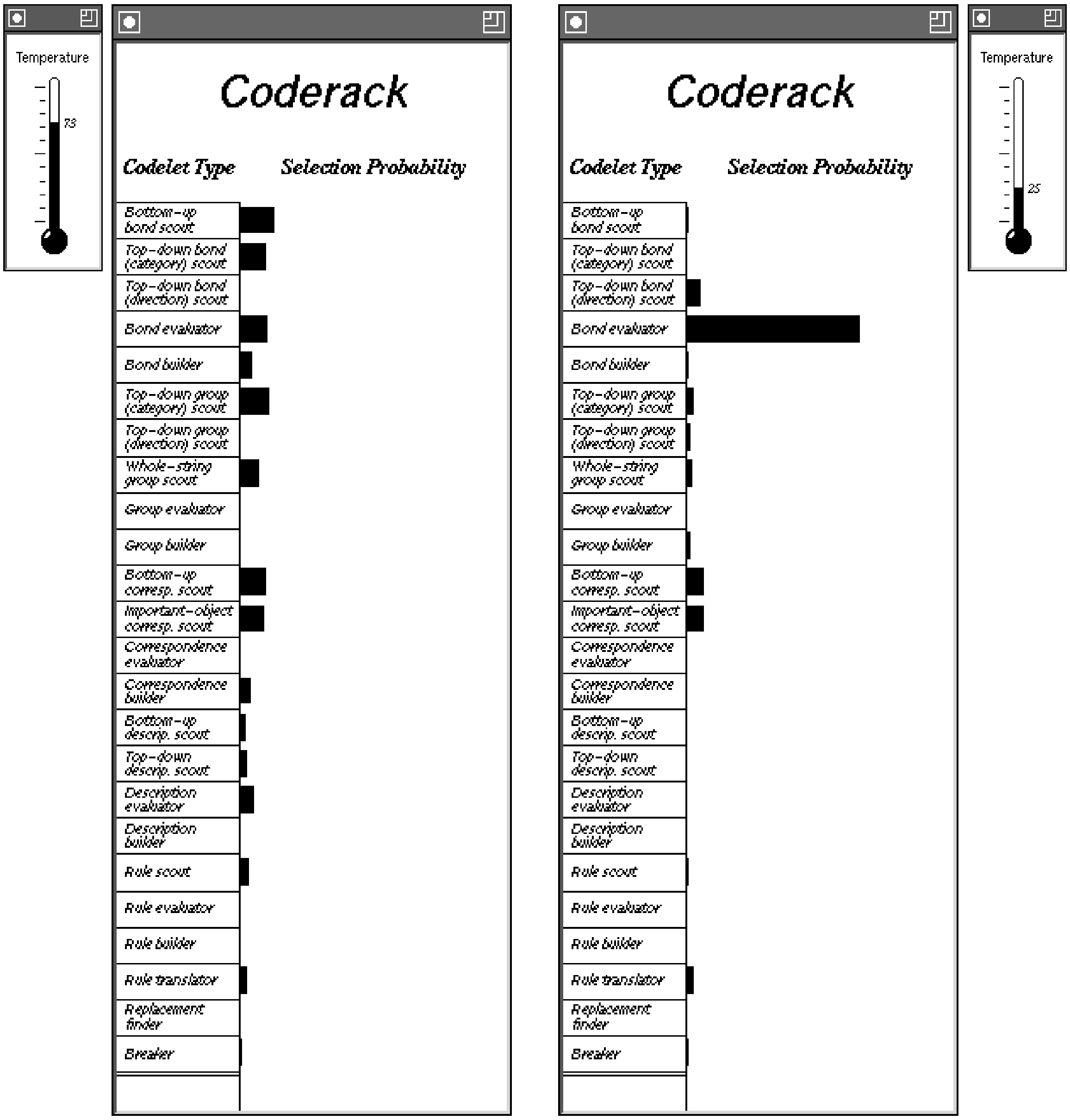


Figure 1.5: *The state of Copycat's Coderack at two different points during a run of the problem "**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**", showing the relative selection probabilities of codelets. The temperature of the Workspace at each point is also shown.*

image shows the situation later in the run, after the temperature has dropped to a much lower value. Processing is now much more serial and deterministic, with the selection of *Bond-evaluator* codelets being strongly favored at the moment—as indicated by the large spike in selection probability. (Typically, at low temperatures, this pattern of spiking varies dramatically from moment to moment—changing according to the need for particular types of codelets to run next. Watching this in "real time" on the screen conveys, in a quite vivid way, the serial nature of processing at low temperatures.)

In summary, Copycat's Workspace temperature guides the program as it explores "interpretation space" in search of strong, consistent mappings between letter-strings. The search for a good configuration of perceptual structures leading to a high-quality answer proceeds via a large number of fine-grained stochastic decisions made by codelets during processing. These decisions, which depend on the current temperature, cause new structures to be built, or existing structures to be destroyed. This changes the temperature, which in turn influences further structure creation, and so on, forming a kind of feedback loop. Temperature thus serves as a very crude mechanism for *self-watching* in Copycat, since it allows the program, to some extent, to regulate its own behavior. That is, by coupling the stochastic activity of codelets to the temperature, the program becomes sensitive to the consequences of its own behavior, since the temperature reflects this behavior in a very broad way. This type of rudimentary self-watching, however, is quite primitive. Accordingly, as will be explained in the next chapter, developing a much more sophisticated approach to self-watching is one of the central goals of the Metacat project.

CHAPTER TWO

From Copycat to Metacat

As a result of the work on Copycat, much light has been shed on many central issues of cognitive science and artificial intelligence, including the nature of concepts and their appropriate representation in computers, the relation of concepts to perception, and the role of emergent computation in computer models of cognition [Hofstadter and FARG, 1995; Mitchell, 1993; Mitchell and Hofstadter, 1990; Mitchell, 1990]. Although this work represents a considerable achievement, it nevertheless must be regarded as only the first step toward a more comprehensive realization of the original ideas underlying the project [Hofstadter, 1984a]. Initial work necessarily concentrated on certain foundational issues of cognition while postponing others for future research. In particular, Copycat focused on the computational modeling of:

- The internal structure of concepts in long-term memory
- The mutual interaction of concepts in long-term memory
- The organization of raw perceptual data into a coherent high-level interpretation
- The interaction between concepts and high-level perception

Coming to grips with these fundamental issues has been the major contribution of the Copycat project.

2.1 A Short History of FARG Work

Copycat is part of a broader ongoing research program whose ultimate objective is to capture as closely as possible, in a computational model, the full range of human psychological processes responsible for high-level perception and analogy-making. This is, of course, a very ambitious goal, given the overwhelming subtlety and complexity of human cognition. Over the years, several projects by Hofstadter and his colleagues in the Fluid Analogies Research Group (FARG) have taken different routes toward this same general goal by focusing on different aspects of high-level perception [Hofstadter and FARG, 1995]. All of these projects have involved building computer models that operate in carefully-designed microworlds.

2.1.1 Jumbo

The earliest such project—and the one most directly related to Copycat—was called Jumbo, and modeled the processes involved in chunking unstructured perceptual parts into hierarchical, integrated wholes [Hofstadter, 1983]. Jumbo worked in the domain of anagram puzzles, attempting to rearrange a given set of "jumbled" letters into English-like words. The program's knowledge was limited to the general clustering properties of vowels and consonants in the English language (*e.g.*, the fact that *s* and *h* form a frequent consonant cluster, while *z* and *q* do not). It had no built-in dictionary of English words to consult. Coming up with actual English words, however, was not the point; the focus of Jumbo was on building *fluid* representational structures—structures that could be easily reconfigured at a moment's notice. Solving word jumbles clearly requires the ability to regroup and reshuffle combinations of letters on many different levels (for example, reperceiving "week-nights" as "wee-knights", or rearranging "pang-loss" into "lang-poss" or "loss-pang"). This type of representational fluidity, however, is not limited solely to anagram puzzles, but is instead a deep property of perception in general. Work on Jumbo focused on developing the computational mechanisms needed to support flexible, malleable perceptual representations. In fact, Jumbo's reconfigurable data structures were the precursors to Copycat's bonds and groups. The Jumbo architecture also pioneered the idea of the parallel terraced scan, and incorporated computational temperature as a crude form of self-watching. Unlike Copycat, however, Jumbo did not attempt to model concepts at all.

2.1.2 Seek-Whence

Another early precursor to Copycat was the Seek-Whence project [Meredith, 1986; Meredith, 1991], which modeled the perception of abstract patterns hidden in open-ended sequences of numbers, such as the one shown below:

2 1 2 2 2 2 2 3 2 2 4 2 2 5 2 2 ...

The task of the program was to try to predict the next number in a sequence. Sequences were presented to the program one term at a time rather than all at once, which required the program to continually refine and, if necessary, revise its understanding of the basis of the sequence as new terms were provided. Unlike many sequence extrapolation programs, Seek-Whence had almost no knowledge of mathematical concepts beyond simple integer predecessorship and successorship; thus, in particular, it had no knowledge of addition, subtraction, or other arithmetical operations. Instead, the strength of the program was its ability to create hierarchical perceptual structures and to reorganize them dynamically according to context as more and more terms of a sequence appeared. After seeing the first five terms of the above sequence, for example, the program might settle on the idea of a simple alternation between *2*'s and the sequence of natural numbers, thus leading it to incorrectly predict a *3* as the next term. In the light of this and subsequent terms, it would be forced to revise its view of the sequence in favor of some other representation. Eventually, after seeing enough terms, it might reperceive the sequence as a progression of integers in which each one is *surrounded* by *2*'s, effectively shifting the perceptual boundaries of the sequence's basic underlying pattern of organization. Work on Seek-Whence thus broadened the development of fluid perceptual mechanisms begun in Jumbo by focusing on the critical notion of context-sensitivity. Like Jumbo, however, Seek-Whence made no attempt to model the structure of concepts themselves.

2.1.3 Tabletop

The fundamental nature of concepts was addressed by Copycat, and by the Tabletop project already mentioned in Chapter 1. Tabletop was an idealized model of analogy-making in a world of objects on a table, such as cups, glasses, and silverware [French, 1995]. A particular object on one side of the table would be singled out, or "touched", and the program's task was to "do the same thing" from the point of view of an observer seated on the other side of the table. Which object was seen as the counterpart to the touched object from the new perspective depended on many factors, including the types of objects on the table, their particular spatial arrangement, and their semantic connotations. Touching a fork on one side of the table, for example, might correspond to touching a spoon on the other side if no fork were available there, since the concepts of *fork* and *spoon* are generally associated quite closely with each other in most people's minds. Many subtle and competing pressures to touch various objects could be created and systematically varied by changing the relative positions and groupings of objects on the table. Like Copycat, Tabletop's perception of a given situation was guided by a context-sensitive network of active concepts. Unlike Copycat, however, Tabletop explored high-level perception in a two-dimensional domain in which spatial proximity played a key role. It also utilized a somewhat different approach to calculating computational temperature. Nevertheless, both models incorporated similar architectural components and processing mechanisms supporting fluid concepts.

2.1.4 Letter Spirit

Finally, the Letter Spirit project extended the fluid conceptual machinery developed in Copycat and Tabletop to the world of visual letter perception and design [McGraw, 1995; Hofstadter and FARG, 1995]. Initial work on this project concentrated on the perception and categorization of *gridletters*, which are highly stylized letterforms of the lowercase roman alphabet drawn on a two-dimensional grid consisting of 56 allowable line segments [Hofstadter, 1987]. Designing a full set of gridletters from *a* to *z* in a single abstract, yet well-defined style is a challenging act of artistic creation. The goal of the Letter Spirit project is to develop a program capable of perceiving the visual style common to an initial set of gridletters, and then designing the rest of the alphabet in the same style. This very ambitious project, currently in its second phase, is intended to model the deepest aspects of creative artistic design.

A key element of the Letter Spirit architecture is the "central feedback loop of creativity", in which the program not only *creates* new letterforms in a particular style, but also *judges the quality* of the letterforms it creates, in order to assess how well they actually reflect the desired style, possibly revising them as a result. This continual cycle of creation, assessment, and revision is essential to the design process, and ought to play a key role in any faithful computer model of creativity. Current work on Letter Spirit is focused on imbuing the program with this type of ability to step back and evaluate its own performance, something almost entirely lacking in Copycat, and is closely related to the central issues of Metacat [Rehling, 1997; Rehling, 1999].

2.2 Copycat's Weaknesses

In many ways, Copycat is a strong, psychologically-plausible model of creative analogy-making. The range of answers it finds on many analogy problems is comparable to the range of answers given by people. Furthermore, the answers most frequently found by the program tend to be the ones most often suggested by people [Mitchell, 1993]. Moreover, Copycat's rankings of its answers according to their final temperature values often agrees quite well with people's intuitive judgments of answer quality.

On the other hand, Copycat sometimes comes up with extremely bizarre answers, based on seeing its strings in ways that a human almost never would. Mitchell identifies three classes of unrealistic answers: (1) *bad-grouping answers*, which result from the program building groups based on no particular motivation, such as building a rightmost group ***rss*** in the string ***ppqqrrss***; (2) *answers involving unmotivated slippages*, in which slippages are made without any underlying motivation; and (3) *answers involving the unmotivated use of group lengths*, in which the concept of group length is seen as playing a role in a problem, but for no particular reason. Fortunately, the program tends to find such answers very infrequently.

Still, the fact that it finds them at all might appear to be a weakness of the program, since this does not seem to accurately reflect human behavior. However, it is actually a strength, because the program's stochastic processing mechanisms keep open the possibility of finding not only "crackpot" answers such as these (albeit infrequently), but also, on occasion, deeply creative answers. No potential way of interpreting the strings is ruled out *a priori*. This is as it should be, provided that Copycat's stochastic mechanisms lead it to find reasonable, run-of-the-mill, human-like answers *most* of the time. Indeed, the fact that Copycat discovers very creative answers infrequently is a strength of the model as well, since a program that almost always discovered deeply creative answers would be no more psychologically plausible than a program that almost always gave nonsensical answers. After all, even the most creative people in the world do not make great, insightful, creative discoveries every day of their lives. The problem with Copycat is not that it sometimes discovers bizarre answers based on random, unmotivated ideas. Rather, the problem is that Copycat does not *recognize* when it has done so. Although Copycat assigns lower temperatures to "better" answers, it does this mechanically and without any insight; it has no *explicit* understanding of what makes an answer good or what makes one nonsensical.

2.2.1 Copycat lacks insight into its own behavior

Copycat's limitations as a general cognitive model become all too apparent when viewed against the wider backdrop of human cognition. Of course, full human cognition is such an extraordinarily complicated phenomenon that no computer model could hope to capture it in its entirety, at least given the current nascent state of cognitive science. The aim of Copycat, however, has always been to model the *essence* of cognition as faithfully as possible by isolating its most important and indispensable features. But here the model suffers from a serious weakness. Stated simply, Copycat has virtually no *insight* into the answers it comes up with. It is unable to say why a particular answer it has found makes sense (or doesn't), or how it arrived at the answer, or how the answer compares to other possible answers. In contrast, people are usually able to give an account of why they consider some Copycat analogies to be better or worse than others. Something of central importance to human cognition is clearly missing from the Copycat model.

The reason for this lack of insight is that Copycat focuses almost exclusively on perceiving patterns and relationships in its *perceptual data* (the letter strings), while ignoring patterns that occur in its own *processing* of those data. It lacks any explicit, internal representation or knowledge of the underlying process that leads to the discovery of an answer—knowledge that could provide a basis for evaluating the answer's strengths or weaknesses, thereby permitting an insightful assessment of its quality. Said another way, the problem is that Copycat's behavior is too *unconscious*. Unlike people, when Copycat solves analogy problems it exhibits an almost complete lack of "awareness" of what it is doing and of the ideas it is working with. Of course, this is not too surprising, since Copycat was intended to be a model of the *subcognitive* mechanisms underlying cognition. All of the nondeterministic codelet activity in the Workspace—the building of bridges and groups, the making of slippages, and so on—was intended to represent perceptual activity carried out at the subcognitive level, below the level of "consciousness". Copycat's lack of a higher cognitive level, however, is a major deficiency of the model, and stands in stark contrast to human cognition, since people are generally aware of their own thought processes, at least on some level.

For example, an interesting psychological phenomenon called the *self-explanation effect* has been described and studied in the context of students learning to solve physics problems from examples [Chi et al., 1989; Chi et al., 1994]. In this series of studies, students mentally monitored their own comprehension or misunderstanding as they studied worked-out textbook examples of mechanics problems, generating verbal explanations of the example solutions in the process. Those students who learned most effectively from the examples were consistently able to generate more detailed and in-depth explanations of their understanding, demonstrating a greater capacity for accurate monitoring of their own cognitive processes, which in turn reduced their reliance on worked-out examples in solving subsequent problems. Such studies clearly illustrate the ability of people to pay attention to patterns in their own thinking. (See also [Pirolli and Bielaczyc, 1989; VanLehn et al., 1992; VanLehn and Jones, 1993; Sandoval et al., 1995].)

As was mentioned at the end of Chapter 1, computational temperature can be viewed as a rudimentary form of self-watching in Copycat. But such a simple feedback mechanism is far too crude to be considered a serious model of self-awareness. Furthermore, although the final temperature of an answer can serve as a rough indication of answer quality, it offers no insight at all into *why* an answer is good or bad. A single integer value simply doesn't contain enough information. In order for the program to gain a deeper level of insight into its answers, it must achieve a deeper understanding of its own behavior. A sophisticated self-watching ability is needed.

An example that makes Copycat's lack of awareness of its own behavior painfully clear is the following analogy problem:

abc ⇒ ***abd***
xyz ⇒ ***?***

In Copycat's microworld, the letter ***a*** has no predecessor and the letter ***z*** has no successor. The alphabet was explicitly designed not to cycle back to ***a*** after ***z***, so an answer such as ***xya***, based on taking the successor of ***z*** in ***xyz***, is impossible. One is forced to adopt a different strategy as a result of this constraint. One way out is simply the literal-minded answer ***xyd***. On the other hand, if the alphabetic symmetry between the "opposite" letters ***a*** and ***z*** is noticed, then the elegant 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, with ***abc*** going to the right via successorship and ***xyz*** going to the left via predecessorship.[1] This answer is quite creative, and most people see ***wyz*** as being strongly analogous to ***abd***, even though the idea is not at all obvious at first.

When Copycat tries to solve this problem, it almost invariably perceives ***abc*** and ***xyz*** as going in the same direction, which is certainly a reasonable thing to do. However, this interpretation of the situation leads inevitably to an attempt

[1]Equivalently, one could see ***abc*** as a left-directed predecessor group and ***xyz*** as a right-directed successorship group, but this doesn't change the symmetry.

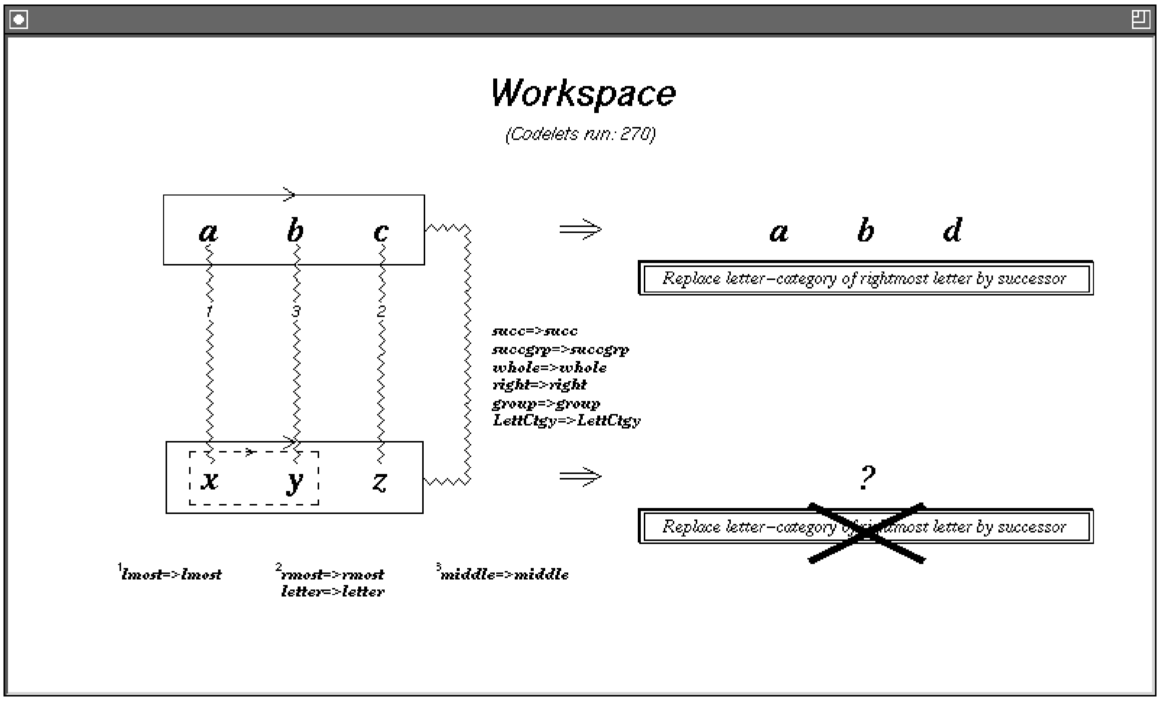


Figure 2.1: *A snag situation resulting from Copycat's attempt to change the letter **z** to its successor.*

to take the successor of ***z***, since ***z*** is seen as corresponding to ***c***. This attempt fails, and Copycat "hits a snag", as shown in Figure 2.1. It is forced to reinterpret the situation. Often, it circumvents this difficulty by changing the rule describing ***abc*** ⇒ ***abd*** from *Replace letter-category of rightmost letter by successor* to *Replace letter-category of rightmost letter by 'd'*, leaving intact the same-direction mapping between ***abc*** and ***xyz***, which then yields the answer ***xyd***. Sometimes, however, the same-direction mapping itself is broken and eventually replaced by a very different, *crosswise* mapping based on the opposite ***a***–***z*** symmetry, yielding ***wyz***. More often than not, though, after breaking the mapping, Copycat tends to rebuild the same structures all over again, which leads it right back to the snag situation. Round and round in circles it goes, hitting the snag over and over again, until it finally manages to stumble onto some other way out, such as falling back on the literal-minded rule mentioned above. Unfortunately, Copycat hits the snag an average of nine times per run on this problem—and sometimes as often as twenty or thirty times in a single run. This is quite unlike typical human behavior. People tend to "get the message" after attempting some unsuccessful strategy a few times. They either get bored and give up completely, or, recognizing that their strategy isn't working, they try something different.

2.2.2 Copycat cannot remember what it has done

As this example makes clear, the program is unable to recognize when it has fallen into a repetitive pattern of behavior. It has no memory of its actions over time, and thus cannot compare its current situation to other situations encountered in the past. This holds true not only on short-term time scales involving a single run of the program in which some situation is encountered over and over again (*i.e.*, a snag), but also on longer time scales involving several answers to a single problem, or different answers to different problems. Once Copycat discovers an answer to a problem, it stops and reports the answer, along with the final temperature, but does not retain this information further. On subsequent runs of the program, no recollection of the answer is possible. This makes comparison of different answers impossible, either for a single analogy problem or among different problems.

Copycat's Workspace and Slipnet are sometimes regarded as the program's short-term and long-term memory. To some extent, this is justified, since the Workspace contains perceptual structures that exist only during the course of a run, whereas the Slipnet contains the permanent set of concepts the program understands about its microworld. This conceptual information is hard-wired into the program, and thus persists over the course of many runs. The activations of Slipnet concepts, however, are reset to a standard initial state at the beginning of every run. Changes in conceptual distances and activation levels that occur during a run are not retained after the program stops with an answer. Likewise, all perceptual structures built in the Workspace during a run are erased as soon as a new run is begun. Consequently, any type of learning that might occur over multiple runs of Copycat is impossible—although, to be fair, learning *per se* was never intended to be a central focus of the project, since the notion of learning to make "better" Copycat analogies is not entirely clear. Nevertheless, it is clear that in order for the program to be able to recognize patterns in its own behavior, it needs a more sophisticated type of short-term memory than what the Workspace provides, and in order for it to be able to compare different answers to a given problem, or to compare different problems *as wholes*, it needs a more comprehensive type of long-term memory than what the Slipnet provides.

2.2.3 Copycat cannot perceive differences between strings

Yet another limitation of the model concerns the creation of rules. Work on Copycat focused on the mapping process between the initial and target strings, and paid relatively little attention to the creation of rules describing the change from the initial string to the modified string. That is, the first phase of development concentrated on developing mechanisms for perceiving *similarity* between strings via bridges and slippages, rather than on characterizing *differences* between strings via rules. For example, in the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"*, Copycat is able to see ***abc*** and ***mrrjjj*** as being "the same" by building a mapping between the strings in which both are represented as successor groups at an abstract level of description, based on concept-mappings such as *Letter-Category* ⇒ *Length* and *successor-group* ⇒ *successor-group*. In contrast, ***abc*** and ***abd*** differ by just a single letter, and this difference is relatively easy to characterize in terms of a rule.

In fact, Copycat places severe restrictions on what types of changes are allowed to the initial string. At most, a single letter-category change involving just one letter is allowed. Thus, the program is able to characterize ***abc*** $\Rightarrow$ ***abd***, but more general changes—even as simple as ***abc*** $\Rightarrow$ ***cba*** or ***abc*** $\Rightarrow$ ***abcc***, in which more than one letter changes or the length of the string changes—cannot be captured by any rule. Under such rigid restrictions, creating an initial/modified mapping and abstracting a rule based on it is an essentially trivial task, because such a mapping is always one-to-one and always involves just one possible type of change. Developing more robust mechanisms for perceiving and characterizing differences between strings was postponed to a later phase of the project.

2.3 The Objectives of the Metacat Project

2.3.1 Handling arbitrary strings

Accordingly, one of the primary objectives of the Metacat project has been to extend and generalize Copycat's rule-building mechanisms so that the program is able to handle a wider class of analogy problems. Restrictions on the types of changes allowed between the initial string and the modified string have been greatly relaxed. To this end, the bridge-building mechanisms for creating mappings between strings, which were fully applied only to the task of perceiving similarity between the initial and target strings in Copycat, have been generalized in Metacat to handle arbitrary mappings between the initial and modified strings. This is an important step toward increasing the model's flexibility, although it does not constitute a major conceptual advance beyond the theoretical ideas originally developed in the Copycat architecture. Nevertheless, developing a generalized ability to perceive similarities and differences between arbitrary strings lays the necessary groundwork for addressing Metacat's further objectives.

Hofstadter has outlined five other important challenges to be addressed in any future work stemming from Copycat [Hofstadter and FARG, 1995, Chapter 7]. For the most part, meeting these challenges involves overcoming the weaknesses of Copycat discussed earlier. The remainder of this section reviews these five objectives and discusses to what extent they have been addressed in the development of the Metacat architecture described in this dissertation. A detailed discussion of Metacat's expanded rule-building mechanisms will be deferred until Chapter 3.

2.3.2 Self-watching

The central, long-term goal of the Copycat line of research is to computationally model how high-level cognitive phenomena such as creativity, self-awareness, and understanding can arise out of a subcognitive substrate composed of a huge number of tiny, nondeterministically-interacting micro-agents, each of which is far too small by itself to support such phenomena. Few people would suggest that individual neurons in the brain (or individual molecules, for that matter) are "conscious" in anything like the normal sense in which humans experience consciousness. Unless one is mystically inclined, one is forced to accept the fact that consciousness arises, somehow, out of nothing but billions of individual molecular 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? Hofstadter argues that two key ideas are of paramount importance [Hofstadter and FARG, 1995, page 311]:

> 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.

Such a capacity for self-monitoring (or *self-watching*) rests on a foundation of high-level perception—already well-developed in Copycat—which allows concepts to be used in very flexible ways. The principal objective of Metacat is thus to develop mechanisms that will allow the program to monitor its own actions and, consequently, to *make explicit* the ideas that come into play during the course of solving a given analogy problem. This amounts to building a higher-level "cognitive" layer on top of Copycat's "subcognitive" layer, which can watch and remember what happens at the lower level as perceptual structures are built, reconfigured, and destroyed in pursuit of an answer.

To do this, Metacat needs to create an explicit sequential record of the most important processing events that occur as it works on a problem. The temporal record left behind by the program can then be examined by codelets for patterns—in much the same way that Copycat's codelets examine letter-strings for patterns. By monitoring its own perceptual processing, and by building explicit representations of this activity, Metacat should be able to achieve a deeper awareness of what its answers are really *about* by examining the key ideas and events that led to the discovery of particular answers. Furthermore, it should be able to recognize when it has fallen into a repetitive or otherwise unproductive pattern of behavior. Recognizing that it is stuck in a rut should enable it to subsequently "jump out of the system" by explicitly focusing on ideas other than the ones that seem to be leading it nowhere.

2.3.3 Episodic memory and reminding

Metacat's ability to create an explicit temporal trace of its "train of thought" should enable the program to form abstract, high-level characterizations of the answers it finds by extracting from the trace the most essential information about an answer. The program should store these abstract answer characterizations in a long-term episodic memory, allowing the program to gradually build up, over the course of several analogy problems, a repertoire of experience on which to draw when confronted with new situations—rather than simply forgetting everything about an answer as soon as a new problem is started. After having seen a number of letter-string analogy problems, people are often reminded of previous problems when confronted with a new problem that is similar to one they have already seen. Likewise, Metacat should sometimes be "reminded" of answers it has previously seen if they are sufficiently similar to an answer just found.

2.3.4 Comparing and contrasting answers

When people come up with more than one answer to an analogy problem, or when they get reminded of some other answer they have encountered before, they can usually explain—if pressed to do so—why the answers seem similar, or how they differ. They are able to compare and contrast answers in terms of the key ideas involved. Metacat should be able to do this as well. For example, the essence of the ***mrrjjjj*** answer to the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ ?"* lies in seeing both ***abc*** and ***mrrjjj*** as successor groups, one based on the idea of letter category and the other based on the idea of group length. This abstract similarity is what fundamentally distinguishes the answer ***mrrjjjj*** from other, more straightforward answers to the same problem, such as ***mrrkkk***, ***mrrjjk***, or ***mrrddd***, all of which overlook the hidden successorship fabric lurking beneath the surface of ***mrrjjj***. Although Copycat can recognize ***mrrjjj*** as a successor group, it cannot point to this idea as being the key to the answer ***mrrjjjj***.

Recognizing that the answers ***mrrkkk*** and ***mrrjjk*** are fundamentally similar in a way that ***mrrkkk*** and ***mrrjjjj*** are not amounts to making analogies between analogies, since each answer itself constitutes an analogy in the world of letter-strings. Thus, an ability to make "meta-analogies" arises naturally from the ability to map answers onto each other, whether through the process of reminding via spontaneous memory retrieval, or through direct comparison of alternative answers to a problem.

2.3.5 Working backwards from a given answer

An ability to insightfully evaluate the relative strengths and weaknesses of different answers should make it possible for Metacat to evaluate not only its own answers, but also answers suggested to it by an outside agent. In other words, Metacat should not only be able to *find* answers to analogy problems, it should also be able to *justify* answers on their own terms, even if the program itself didn't come up with them. This amounts to "working backwards" from a given answer toward an insightful characterization of the answer, in order to understand why it makes sense. Once an answer has been understood in this way, it can be compared and contrasted with other answers that the program has either discovered previously itself, or been shown by someone else.

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 weren't able to think of it themselves. In a similar vein, suggesting the somewhat "tongue-in-cheek" answer ***abd*** usually elicits a few chuckles from people, indicating that they can see how it "makes sense", although practically no one gives this answer on their own [Mitchell, 1993]. Of course, 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.

2.3.6 Making up new analogy problems

Finally, a very long-term goal of the Metacat project is to endow the program with the ability to make up entirely new, high-quality analogy problems on its own. This would require Metacat to have not only a deep understanding of the issues that arise in individual analogy problems, but also a deep grasp of the inherent subtleties of letter-string analogies in general. Such an ability would represent the program's attainment of expert-level mastery over its microdomain—something that requires a great deal of experience even for humans to attain. Inventing elegant and clever letter-string analogy problems is a skill that is only acquired through *doing* many analogy problems and by being acutely aware of the various competing pressures evoked by rival answers. Thus, in any program able to invent its own problems, *learning from experience* would almost certainly have to play a critical role. As the program invented more and more analogy problems, it would gradually learn what makes for interesting problems, and the subtlety and sophistication of its creations would increase. In order to do this, the program would have to be able to "try out" problems it had invented, comparing and contrasting the various possible answers with one another and with other similar problems stored in its memory, and to recognize when it had come up with something intriguing.

For example, the best problems are often those in which a straightforward, easy-to-see answer masks a more elegant answer hidden "below the surface", such as the problem *"**apc** ⇒ **abc**; **opc** ⇒ **?**"* [Hofstadter and FARG, 1995, page 317]. For this problem, one possibility is to "take the bait" offered by the ***pc*** in both ***apc*** and ***opc*** and simply change the ***p*** to ***b***, obtaining ***obc***. On the other hand, a more abstract way of looking at things yields the answer ***opq***, in which ***apc*** ⇒ ***abc*** and ***opc*** ⇒ ***opq*** are both seen as "fixing a one-letter flaw" in successor groups of length three. Recognizing this problem as being interesting would require the program to appreciate the delicate interplay of pressures between the rival answers ***obc*** and ***opq***.

Once the program had pinpointed the key pressures inherent in a particular problem, it might then go on to suggest subtle variants of the problem by systematically changing the balance of pressures in interesting ways, possibly creating an entire family of related problems stemming from the original. Such an ability of the program to step back and assess the quality of its own analogy-problem creations, possibly revising and improving them as a result, is very similar in flavor to the "central feedback loop of creativity" of the Letter Spirit project mentioned earlier in section 2.1.4.

2.3.7 The objectives of the present work

All of the above objectives except for the last one (making up new analogy problems) have been addressed in the development of the Metacat architecture described in this dissertation. At the present stage, I believe that it is too early to attempt to imbue the program with even a rudimentary ability to make up its own problems. A program able to consistently invent good analogy problems would require a qualitatively different level of understanding than that modeled by the current Metacat program. Such an escalation in ability would be, I believe, comparable in magnitude to the escalation from the ability to *make* analogies, as was modeled in Copycat, to the ability to *comprehend* analogies, as is modeled in Metacat. Therefore, tackling this challenge is best left to a later stage of development. Only after the notion of comprehending individual analogies has been thoroughly explored and computationally modeled should the task of building a program able to comprehend *analogy-making in general* be undertaken.

2.4 An Overview of the Metacat Architecture

The remainder of this chapter presents a broad overview of the Metacat architecture, and concludes with a discussion of Metacat's relationship to other work in AI and cognitive science, particularly work in case-based reasoning and derivational analogy.

Metacat is an extension of the Copycat model—not an alternative model designed to supplant it. Consequently, Metacat's architecture includes all of Copycat's main architectural components, such as the Workspace, the Slipnet, and the mechanisms that support distributed, nondeterministic codelet processing. In addition, the mechanisms for building bridges and creating rules have been greatly extended and generalized, and new architectural components have been incorporated into the model. Together, these components provide a common framework in which to address self-watching and the other objectives outlined in the previous section.

2.4.1 The Episodic Memory

Unlike Copycat, Metacat incorporates an episodic memory for its answers, which allows it to remember its problem-solving experiences over time. When the program discovers a new answer, it pauses temporarily to display the answer along with the groups, bridges, rules, slippages, and other Workspace structures that gave rise to it, instead of simply quitting. Together, these structures represent a way of interpreting the analogy problem that yields the answer just found. All of this information, including the problem itself, is then packaged together into an *answer description* and stored in Metacat's memory, after which the program continues searching for alternative answers to the problem. Gradually, over time, a series of answer descriptions accumulates in memory, each one containing much more information than just the answer string itself.

The most important information stored in answer descriptions consists of structures called *themes*. Themes are high-level structures that represent the key ideas underlying an answer. In Copycat, at the time an answer is found, the configuration of structures in the Workspace collectively represents a particular way of interpreting the problem, but which aspects of that interpretation are essential and which are not remains unclear. In Metacat, themes are used to identify and explicitly represent those aspects of the interpretation that are most important in characterizing the answer. This amounts to abstracting out an explicit, high-level description of the answer's *essence* from the many Workspace structures and events that led to its discovery. This high-level answer description is represented as a collection of themes, and serves as the basis for comparing and contrasting the answer to other answers stored in memory. Furthermore, Metacat may get reminded of similar answers it has encountered in the past if the themes associated with a newly-discovered answer, acting as memory retrieval cues, match those of some previously stored answer description sufficiently well. The pattern of themes in an answer description serves as an index under which an answer can be stored and retrieved from memory.

2.4.2 The Themespace

Themes reside in Metacat's *Themespace*, and consist of pairs of Slipnet concepts. For example, a theme representing the idea of alphabetic-position symmetry between two objects is composed of the concepts *Alphabetic-Position* and *opposite*. In some ways, themes act like ordinary Workspace structures. They are not initially present in the Themespace; rather, they get built during the course of a run, in response to the creation of various structures in the Workspace. In the same way that Workspace structures such as bridges and groups explicitly represent patterns among *input letters*, themes explicitly represent patterns among *Workspace structures*. Thus they are in some sense "meta-level" Workspace structures.

In other ways, however, themes act like Slipnet concepts. They can take on various levels of activation, depending on the extent to which the ideas they represent are present or absent in the current configuration of structures in the Workspace. Although each theme consists of a pair of Slipnet concepts, a theme's activation level is distinct from the individual activations of its constituent concepts. Furthermore, a theme's activation may decay over time, and may be influenced by the activation levels of other themes. Like Slipnet concepts, themes can, under certain conditions, exert strong *top-down pressure* on perceptual activity occurring in the Workspace. In fact, themes can assume both positive and negative levels of activation, ranging from -100 to $+100$. Positively-activated themes exert "positive thematic pressure", encouraging the building of Workspace structures compatible with the themes. Negatively-activated themes, on the other hand, exert "negative thematic pressure". Their effect is to *discourage* the creation of compatible structures, promoting instead the creation of structures *incompatible* with the themes.

2.4.3 The Temporal Trace

In addition to the Themespace and Episodic Memory, Metacat's architecture includes a separate short-term memory called the *Temporal Trace* (or just the *Trace* for short) that serves as the focus for self-watching. Like the Themespace, the Temporal Trace accumulates information over the course of a single run, so it can be viewed as an extension of the Workspace. Specifically, the Trace stores an explicit temporal record of the most important processing *events* that occur during the course of solving an analogy problem. Examples of such events include recognizing some key idea pertaining to the problem (by noticing the strong activation of a theme or concept embodying the idea), encountering a snag situation, or discovering a new answer. Of course, a huge number of events of all "sizes" occur during the processing of almost any analogy problem, ranging from fine-grained "micro events" (such as proposing a bond between two letters, or evaluating the strength of some proposed structure) all the way to global "macro events" (such as hitting a snag). 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.

Once events have been explicitly represented in the Trace, they are themselves subject to examination by codelets. This allows Metacat to perceive patterns in its own processing in much the same way that Copycat perceives patterns in its letter-strings—via codelets looking for relationships among perceptual structures. In Copycat's case, these perceptual structures are the letters, groups, bonds, and so on, stored in the Workspace; in Metacat's case, they also include "reified" structures stored in the Trace which are created from events that occur during the processing of Workspace structures. When a new answer is found, an answer description can be formed by examining the temporal record in the Trace to see which events contributed to the answer's discovery.

Figure 2.2 shows a schematic diagram that summarizes Metacat's main architectural components and the principal ways in which they interact. Metacat's basic repertoire of concepts relating to the letter-string microworld are stored in the Slipnet. The Themespace contains aggregate structures (*i.e.*, themes) which characterize, at an abstract level of description, the activity occurring in the Workspace. These structures are themselves collections of Slipnet concepts, and (like Slipnet concepts) can exert strong top-down pressure on processing in the Workspace. This processing, in turn, influences the activation levels of both themes and concepts. Sitting above the subcognitive processing level is the Temporal Trace, which "watches" the activity occurring at the lower levels and records the most important events that take place. Codelets can examine the resulting chain of structures in the Trace, which, under certain circumstances, may result in particular patterns of themes in the Themespace being clamped at high activation. This will in turn strongly influence subsequent activity at the subcognitive level. Once an answer is found, a high-level description of the answer can be formed by extracting from the temporal record the most important themes and events that contributed to its discovery. This description is then stored as a new episode in memory indexed under the appropriate themes. In the

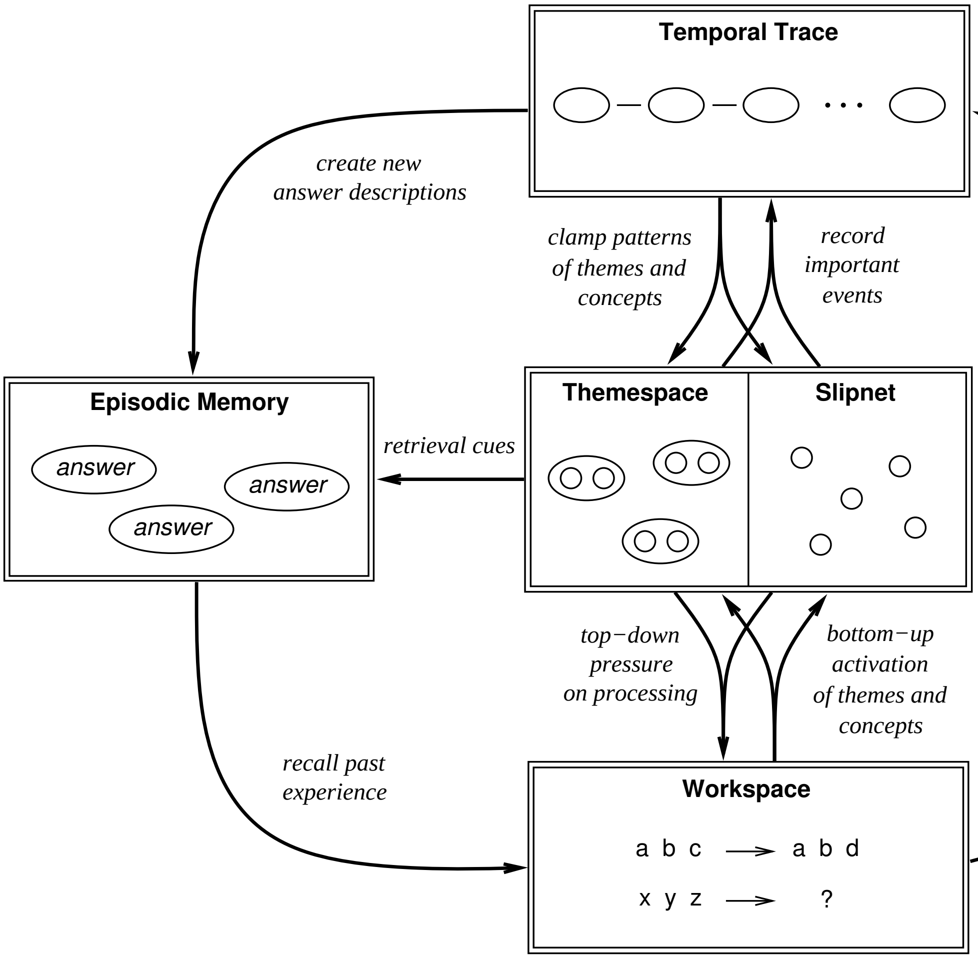


Figure 2.2: *Metacat's main architectural components, and the principal ways in which they interact. The Themespace shares properties of both the Slipnet and the Workspace. Both the Temporal Trace and the Themespace can be regarded as extensions of the Workspace that contain "meta-level" perceptual structures. Together, these three components constitute Metacat's short-term memory. The Episodic Memory and the Slipnet together constitute Metacat's long-term memory.*

future, should Metacat encounter a problem that evokes a pattern of themes similar to the pattern of themes stored with the answer description, the stored episode may be recalled as a result. Furthermore, different answers can be compared to each other on the basis of the information stored in their descriptions.

2.4.4 Themes and self-watching: An example

As was mentioned earlier, themes take on varying levels of activation during the course of processing. At any given moment, a theme's activation level represents an estimate of the importance of its role in characterizing the program's understanding of the situation at hand. Thus, themes are first and foremost representational structures (in this sense they are like Workspace structures). But under certain conditions, when highly activated, they can also exert powerful top-down pressure on Metacat's subcognitive processing mechanisms, strongly biasing the stochastic behavior of codelets in favor of particular outcomes (in this sense they also exhibit Slipnet-like qualities).

As an example, consider again the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"* discussed at the end of section 2.2.1. In Copycat, the idea of perceiving ***abc*** and ***xyz*** as going in the same direction can be represented *implicitly* by a mapping consisting of the vertical bridges ***a***–***x***, ***b***–***y***, ***c***–***z***, and a higher-level bridge between the successor groups (or predecessor groups) ***abc*** and ***xyz***. In Metacat, this state of affairs can be represented *explicitly* by a "vertical" theme based on the concepts of *String-Position* and *identity*, which captures the essential idea underlying the vertical mapping between ***abc*** and ***xyz*** that objects having identical positions in their respective strings correspond to one another. Conversely, seeing the strings as going in opposite directions is represented in Copycat by a "crosswise" vertical mapping involving the bridges ***a***–***z*** and ***c***–***x***, the essence of which can be captured abstractly in Metacat by a *String-Position: opposite* vertical theme.

Should a *String-Position: opposite* theme become highly activated, it will strongly promote the creation of Workspace structures that support the idea of mapping ***abc*** onto ***xyz*** in a "mirror image" fashion, and will suppress the creation of structures incompatible with this idea. For instance, the creation of vertical bridges based on the concept-mappings *rightmost* ⇒ *leftmost* or *leftmost* ⇒ *rightmost* will become extremely likely, whereas *leftmost* ⇒ *leftmost* or *rightmost* ⇒ *rightmost* bridges will be inhibited. Active themes can be thought of as Metacat's way of "seizing on" certain key ideas implicit in an analogy problem and making them explicit, driving the program toward an interpretation of the problem organized around these ideas. Different configurations of active themes in the Themespace will guide Metacat toward different interpretations of an analogy problem, which consequently may cause different answers to be discovered for the problem.

On the other hand, themes can sometimes acquire negative activation. Negatively-activated themes exert negative thematic pressure on codelet processing, which tends to drive the program *away* from certain interpretations of a problem. For example, a strongly-negative *String-Position: identity* vertical theme will discourage the creation of bridges between letters of ***abc*** and ***xyz*** having the same string position, such as ***a*** and ***x*** or ***c*** and ***z***, making it difficult to build a same-direction vertical mapping between the strings. This will in turn push the program into other regions of "interpretation space", encouraging it to explore alternative ways of creating this mapping.

The ability to steer away from certain interpretations of an analogy problem by negatively activating certain patterns of themes offers a way for Metacat to avoid falling into mindlessly repetitive patterns of behavior, or at least to be able to "jump out of the system" when it does end up falling into one. As was seen earlier, Copycat is plagued by such "loopy" behavior on certain problems, for it has no way of noticing patterns in its own processing. Although theme activation is not a sufficient mechanism, by itself, for overcoming repetitive behavior, Metacat's themes nevertheless provide a general framework in which to address this problem, which will now be explained.

When an event is added to the Temporal Trace during the processing of an analogy problem, the themes most active at the time of the event are also noted along with it. These themes serve as the event's *thematic characterization.* In the case of an "answer event", a high-level answer description is abstracted from these themes and the other events in the Trace and stored in Metacat's episodic memory, as was mentioned earlier. For a "snag event", however, the thematic characterization represents a way of interpreting the analogy problem that has just led to failure. If Metacat continues to hit the same snag several times in succession, a series of failure events will accumulate in the Trace, all with very similar thematic characterizations. But since these processing events are now represented as concrete perceptual structures, they are subject to examination and manipulation by codelets in a natural way. Similarity between multiple failure events in the Trace can be noticed by codelets in much the same fashion that similarity between multiple letters or groups in the Workspace is noticed. By monitoring its own behavior in this way, Metacat can recognize when it is stuck in an ongoing, repetitive pattern of behavior. Furthermore, once the program has recognized a particular thematic configuration as leading to failure, it can clamp the "offending" themes with strong negative activation, effectively steering itself away from the unproductive interpretation leading to the snag. In this way, Metacat can both recognize and subsequently break out of repetitive behavior. Figure 2.3 shows this idea schematically.

2.4.5 Working backwards: An example

While negative thematic pressure is useful for avoiding problematic interpretations that lead repeatedly to failure, positive thematic pressure can guide Metacat *toward* interpretations based on particular key ideas. This makes it possible for the program

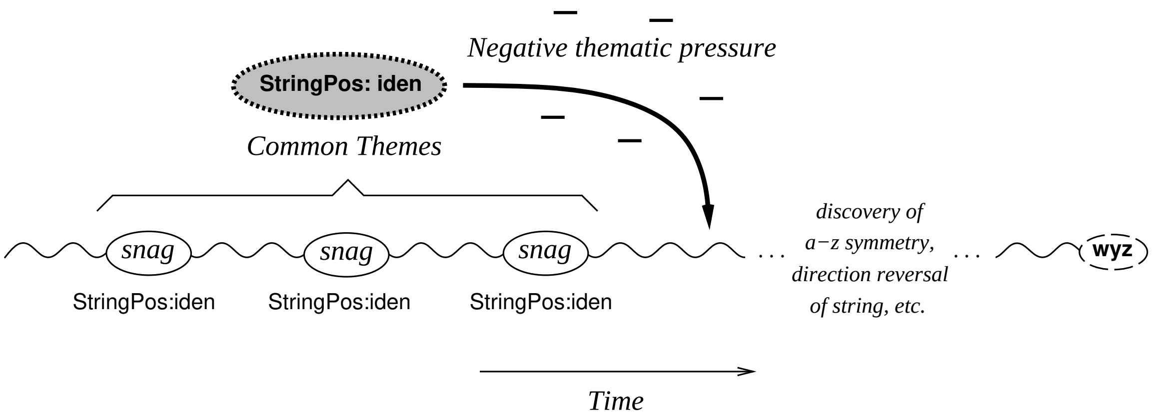


Figure 2.3: *Schematic diagram showing how themes common to several snag events in the Temporal Trace can trigger negative thematic pressure, subsequently steering Metacat away from a problematic interpretation of a problem (e.g., viewing* ***abc*** *and* ***xyz*** *as going in the same direction), and eventually toward alternative interpretations.*

to effectively work backwards on analogy problems, starting from a given answer. When Metacat runs in "justify mode", it takes a problem together with an answer supplied by the user and attempts to discover a way of interpreting the problem in which 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 different parts of this interpretation may suggest new ideas to try out. Metacat can explicitly focus on these ideas, represented as patterns of themes, by clamping the themes with strong positive activation. The resulting thematic pressure forces the program to reorganize its interpretation of the problem in accordance with these ideas, leading to a new—and perhaps more coherent—way of looking at things.

For example, when Metacat is asked to justify the answer ***wyz*** to the problem "***abc*** ⇒ ***abd***; ***xyz*** ⇒ ***?***", it typically begins by building straightforward mappings in which all the strings are seen as going in the same direction. In addition, it may

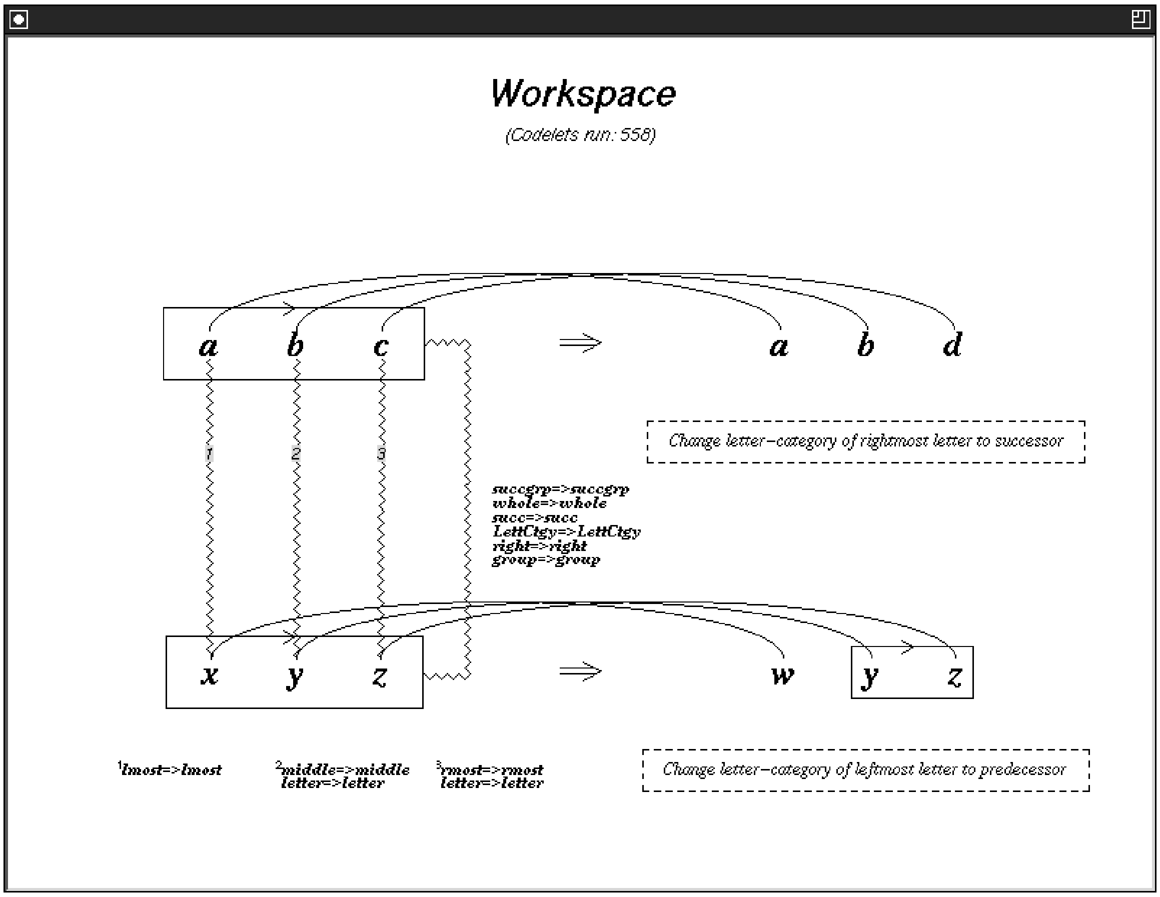


Figure 2.4: *An inconsistent interpretation of the answer **wyz**.*

create a "top" rule describing the ***abc*** ⇒ ***abd*** mapping as *Change letter-category of rightmost letter to successor* and a "bottom" rule describing the ***xyz*** ⇒ ***wyz*** mapping as *Change letter-category of leftmost letter to predecessor*. This state of affairs is shown in Figure 2.4. Although each of the three string mappings making up this interpretation is 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 (since there is no bridge between them), 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 set of vertical themes such as *String-Position: opposite*, *Direction: opposite*, and *Group-Category: opposite*. Metacat can explore the ramifications of this idea by clamping the associated themes at full activation in the Themespace. The resulting positive thematic pressure strongly promotes the creation of new structures compatible with the idea of mapping ***abc*** and ***xyz*** onto each other in a crosswise fashion, and significantly weakens existing structures incompatible with this idea, such as the ***a***–***x*** and ***c***–***z*** bridges. The net effect is that the original vertical mapping shown in Figure 2.4 is swiftly reorganized by codelets into a new mapping consistent with the activated themes. Figure 2.5 shows the final, globally consistent interpretation, in which ***c*** and ***x*** are seen as corresponding. In addition, the previously unnoticed alphabetic-position symmetry between the letters ***a*** and ***z*** has been identified as a result of the increased attention focused on these objects by top-down thematic pressure. Consequently, the final abstract characterization of ***wyz*** includes an *Alphabetic-Position: opposite* vertical theme based on the *first* ⇒ *last* slippage underlying the ***a***–***z*** bridge.

2.4.6 Comparing and contrasting answers: An example

As the previous example illustrates, themes allow Metacat to "size up" answers that are suggested to it by others, by working backwards to discover interpretations that make sense. For answers that the program discovers on its own, it can examine the history of events in the Trace in order to create abstract descriptions of the answers. In either case, once answers have been described in terms of their key underlying themes and stored in memory, they can be compared and contrasted with each other on the basis of these descriptions.

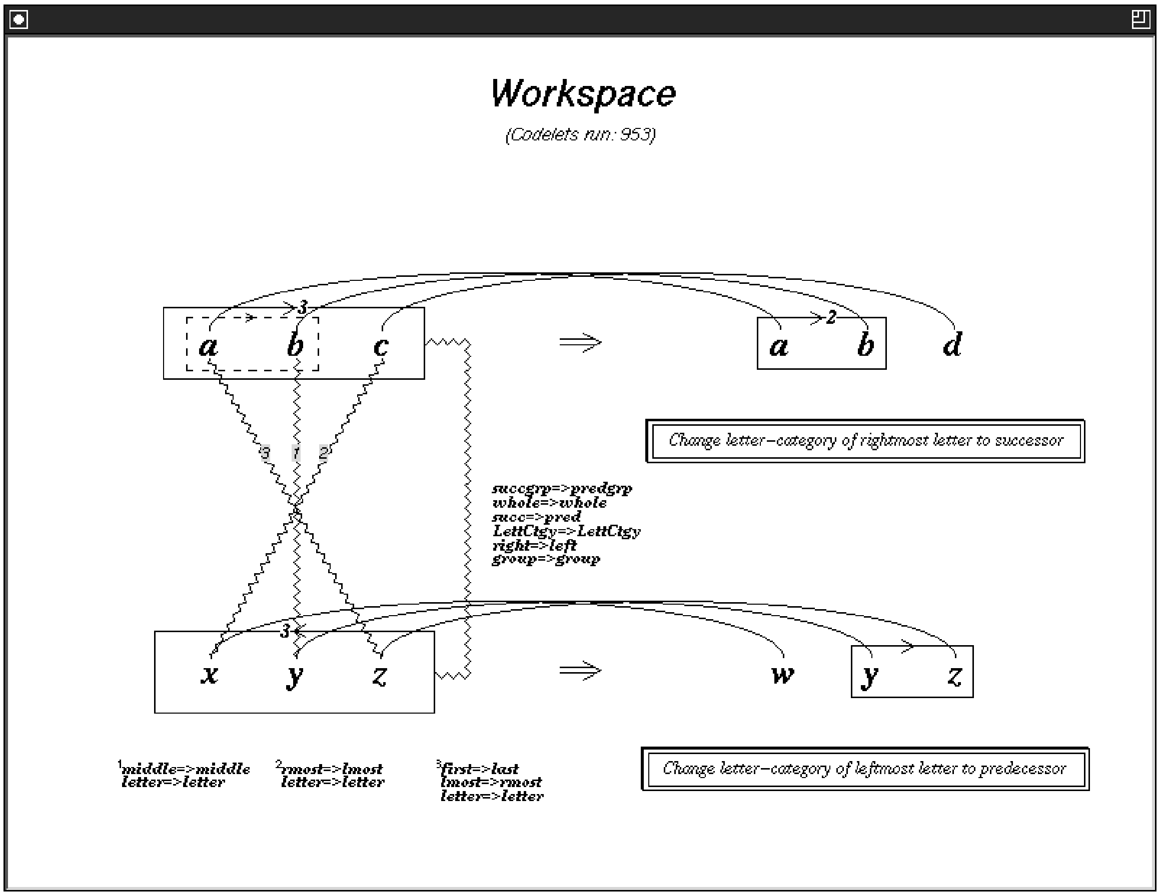


Figure 2.5: *The final consistent interpretation of* ***wyz****.*

As an example of how the similarities and differences between analogy problems can be understood in terms of themes, consider again the answer ***wyz*** just described for the problem *"**abc** ⇒ **abd**; **xyz** ⇒ ?"* (Figure 2.5). This answer relies on an interpretation of the problem in which ***abc*** and ***xyz*** are seen as going in opposite directions, ***abc*** and ***abd*** are seen as going in the same direction, and ***abc*** is seen as changing to ***abd*** in an abstract way rather than in a more literal-minded fashion (*i.e.*, this change is described by the rule *Change letter-category of rightmost letter to successor*). At the crux of this interpretation lies the alphabetic-position symmetry of the letters ***a*** and ***z***, which provides the justification for perceiving ***abc*** and ***xyz*** as "mirror images" of each other. These ideas can be represented abstractly by a collection of structures that includes *Alphabetic-Position: opposite* and *String-Position: opposite* vertical themes, a *String-Position: identity* top theme, and the aforementioned rule. Together, these structures constitute ***wyz***'s answer description in long-term memory.

In contrast, Figure 2.6 shows Metacat's Workspace after it has found the answer ***xyd***. In this interpretation of the problem, ***abc*** and ***xyz*** are seen as going in the same direction, with letters in identical string positions linked by vertical bridges. The strings ***abc*** and ***abd*** are mapped onto each other in a similar fashion, as shown by the horizontal bridges across the top, and the ***c*** in ***abc*** is seen as changing literally to ***d***, as indicated by the rule *Change letter-category of rightmost letter to 'd'.* These are the essential ingredients of the answer ***xyd***, and they can be explicitly represented by an answer description that includes a *String-Position: identity* vertical theme, a *String-Position: identity* top theme, and the above rule. The idea of alphabetic-position symmetry does not arise in the case of ***xyd***, so there is no corresponding *Alphabetic-Position: opposite* theme involved.

Now consider the problem *"**rst** ⇒ **rsu**; **xyz** ⇒ ?"*, which is similar in many respects to the *"**abc** ⇒ **abd**; **xyz** ⇒ ?"* problem. In particular, the answers ***xyu*** and

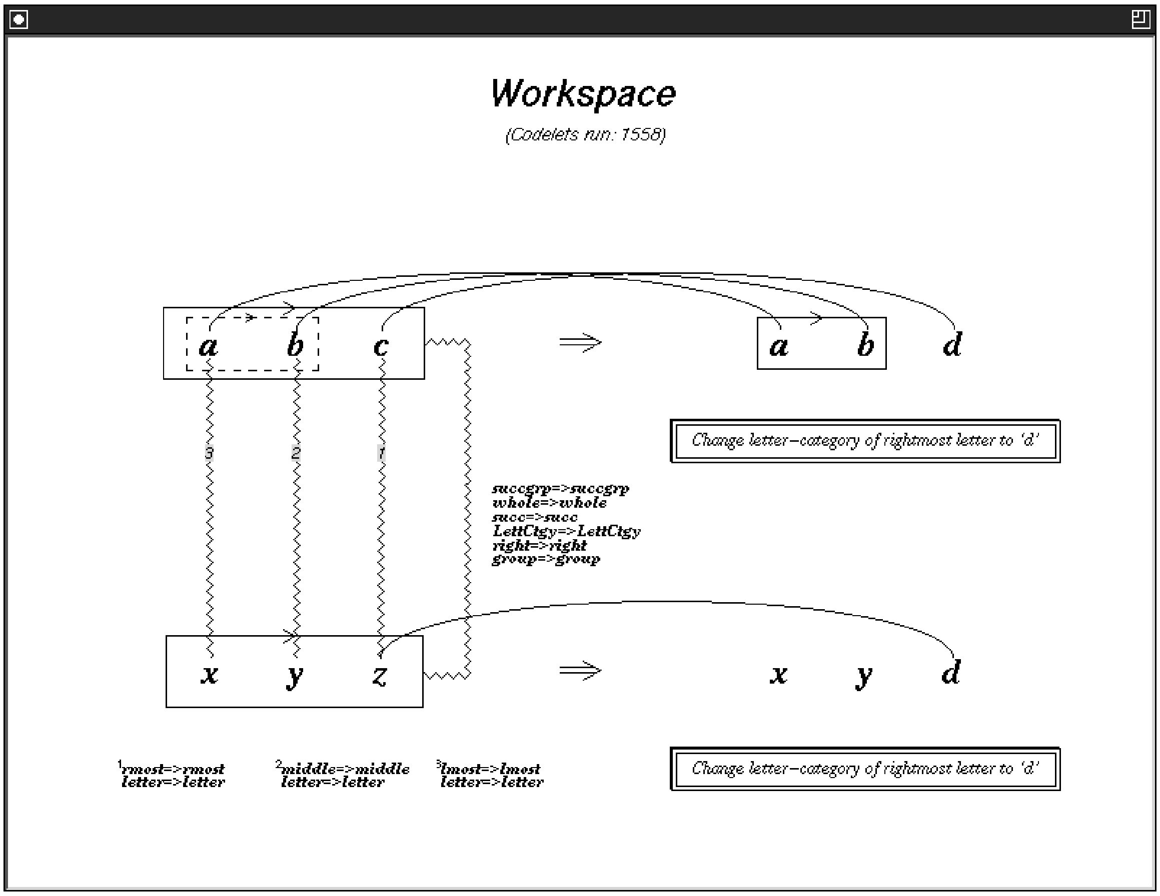


Figure 2.6: *An interpretation of the problem "**abc**⇒ **abd**; **xyz**⇒ **?**" that yields the answer **xyd**, showing the various Workspace structures involved.*

wyz are possible, based on many of the same considerations that applied in the earlier problem. Seeing the answer ***xyu*** rests in part on seeing ***rst*** and ***xyz*** as going in the *same* direction, while the answer ***wyz*** depends on seeing these strings as going in *opposite* directions. However, in this problem there is far less justification for seeing ***rst*** and ***xyz*** as mirror images of each other, unlike in the previous case of ***abc*** and ***xyz***, with their strong ***a***–***z*** symmetry. Indeed, the presence or absence of alphabetic-position symmetry is the crucial difference between the two ***wyz*** answers. Everything else about them is the same: both involve seeing ***abc*** and ***xyz*** (or ***rst*** and ***xyz***) as going in opposite directions, both involve seeing ***abc*** and ***abd*** (or ***rst*** and ***rsu***) as going in the same direction, and both involve viewing the ***abc*** ⇒ ***abd*** (or ***rst*** ⇒ ***rsu***) change abstractly rather than literally. The diminished justification for the answer ***wyz*** in this problem tends to diminish its overall quality. While arguably better than ***xyu***, ***wyz*** is not nearly as superior to ***xyu*** as was ***wyz*** to ***xyd*** in the previous problem. In short, ***xyd*** and ***xyu*** play essentially *identical* roles in their respective problems, and are thus of comparable quality, while the two ***wyz*** answers are quite *different*, even though on the surface they appear to be identical.

In addition to these four answers, there are two other possibilities worth mentioning. The answer ***dyz***, although perhaps a bit far-fetched, is certainly possible for the problem *"**abc** ⇒ **abd**; **xyz** ⇒ ?"*. Seeing this answer depends on noticing the abstract alphabetic symmetry between ***abc*** and ***xyz***, and yet—somewhat ironically—taking a very literal-minded view of the way in which ***c*** changes to ***d***. Thus, making the "analogous" change to ***xyz*** involves changing its leftmost letter simply to ***d***. The answer ***uyz*** for the problem *"**rst** ⇒ **rsu**; **xyz** ⇒ ?"* arises in a similar manner, except that here there is no good reason to see ***rst*** and ***xyz*** as mirror images of each other in the first place. Just as for the two ***wyz*** answers, the key difference between ***dyz*** and ***uyz*** lies in the presence or absence of the idea of alphabetic-position symmetry. In other words, the *way* in which the two ***wyz*** answers are analogous to each other

Problem/Answer	Vertical Themes	Rule Type
abc ⇒ ***abd***; ***xyz*** ⇒ ***wyz***	*Alphabetic-Position: opposite* *String-Position: opposite*	Abstract
rst ⇒ ***rsu***; ***xyz*** ⇒ ***wyz***	*String-Position: opposite*	Abstract
abc ⇒ ***abd***; ***xyz*** ⇒ ***xyd***	*String-Position: identity*	Literal
rst ⇒ ***rsu***; ***xyz*** ⇒ ***xyu***	*String-Position: identity*	Literal
abc ⇒ ***abd***; ***xyz*** ⇒ ***dyz***	*Alphabetic-Position: opposite* *String-Position: opposite*	Literal
rst ⇒ ***rsu***; ***xyz*** ⇒ ***uyz***	*String-Position: opposite*	Literal

Table 2.1: *Six answers and their associated answer descriptions.*

is exactly like the way in which the ***dyz*** and ***uyz*** answers are analogous to each other. Here we have a simple example of a "meta-level" analogy in the letter-string microworld.

Table 2.1 shows these six answers along with their associated answer descriptions.[2] These descriptions bring out very clearly the similarities and differences among the various possible answers to the two problems. For example, it is clear from examining the themes that the crucial distinction between the first ***wyz*** answer and ***dyz*** is whether the ***abc*** ⇒ ***abd*** change is perceived abstractly or literally (as indicated by the rule involved). The thematic characterizations of ***xyd*** and ***xyu*** are identical, revealing the deep underlying similarity between these two literal-minded answers. The difference between the two ***wyz*** answers rests on the presence or absence of the idea of alphabetic-position oppositeness. Furthermore, the *way* in which these answers differ is precisely the same as the way in which ***dyz*** differs from ***uyz***.

This example gives the flavor of how Metacat's characterizations of its answers in

[2]For the sake of clarity, not all of the information stored in these descriptions is shown here. In particular, top themes are also present, as are other vertical themes based on *Direction* and *Group-Type*. In addition to themes, the rules responsible for each answer are also included. Nevertheless, the information shown here captures the essential similarities and differences that exist between the answers.

terms of themes allow it to compare and contrast idealized analogies in an insightful way. Such an ability lies far beyond that of Copycat, which has only a crude numerical measure of "quality" available as a basis for answer comparison. In addition, Metacat's answers can be retrieved from memory on the basis of their stored descriptions. For example, suppose that Metacat finds the answer ***xyu*** to the problem *"**rst** ⇒ **rsu**; **xyz** ⇒ ?"*. If it has previously encountered the answer ***xyd*** to the problem *"**abc** ⇒ **abd**; **xyz** ⇒ ?"*, finding ***xyu*** may remind it of the ***xyd*** answer it has already seen—based on the strong similarity between the themes characterizing ***xyu*** and the stored description of ***xyd***—prompting Metacat to "comment" on the similarity between the two answers.

Metacat's mechanisms that enable it to compare and contrast its answers in this way will be explained more thoroughly in Chapter 4, and detailed sample runs of the program on several families of analogy problems—including the examples discussed above—will be presented and described in Chapter 5.

2.5 Relation to Other Work

Given the centrality of self-awareness to cognition, it is surprising how little work has been done in AI and cognitive science on developing computer models that focus directly on the issue of self-watching. Several researchers, however, have developed models that exhibit some of the flavor of Metacat and Copycat—namely, in their focus on the notion of processing as the emergent consequence of many nondeterministic micro-actions occurring in parallel, and on the idea of spreading activation among nodes of a semantic network in response to context-dependent pressures. The DUAL cognitive architecture, developed by Boicho Kokinov and incorporated into the AMBR model of analogy-making by Kokinov and Alexander Petrov, is one such model [Kokinov et al., 1996; Kokinov, 1994b; Kokinov, 1994a]. In addition, much work has been done on other issues that figure prominently in Metacat, such as analogy-making and reminding (see, for example, [Holyoak et al., 1998; Lange and Wharton, 1994]).

Prior work on analogy-making includes the well-known structure-mapping theory of Dedre Gentner [Gentner, 1983; Gentner, 1989], and the instantiation of this theory in the form of the Structure-Mapping Engine (SME) computer model [Falkenhainer et al., 1990; Forbus et al., 1994]. In a similar fashion, Keith Holyoak and Paul Thagard's multiconstraint theory of analogy has been used as the basis of their ACME computer model of analogy-making [Holyoak and Thagard, 1995; Holyoak and Thagard, 1989]. Computer models of reminding include the MAC/FAC model developed by Ken Forbus, Keith Law, and Dedre Gentner, and Holyoak and Thagard's ARCS model [Thagard et al., 1990; Forbus et al., 1995; Law et al., 1994]. In addition, much pioneering work on memory organization and retrieval has been done by Roger Schank and his colleagues [Schank, 1982]. Over the years, this work has gradually evolved into the now-thriving field of case-based reasoning (CBR).

In particular, Metacat touches on many of the issues underlying research in case-based reasoning (for good overviews of CBR, see [Leake, 1996] or [Kolodner, 1993]). Answer descriptions stored in Metacat's memory can be likened to "cases" in CBR, in the sense that they form a corpus of experience on which the program can draw when faced with new situations. When Metacat finds a new answer, its stored experiences may cause it to be reminded of similar answers it has seen in the past, in a way that is reminiscent of the retrieval of previously-stored cases from memory in CBR according to their degree of similarity to the current situation. The retrieved answer can then be compared to the current answer on the basis of the thematic information stored with it. This is roughly akin to comparing two cases in CBR in order to see how the cases are similar (*i.e.*, which aspects of the stored case can be applied directly, without modification, to the current situation), and how the cases differ (*i.e.*, which aspects must be adapted to fit the new situation). Furthermore, some work in CBR is beginning to address "metacognitive" issues such as introspective reasoning and self-questioning (see, for example, [Oehlmann, 1995; Oehlmann et al., 1994; Ram and Cox, 1994; Fox and Leake, 1994]).

However, there are important differences between CBR and Metacat. First of all, even though Metacat is concerned with solving analogy problems, it is not intended to model problem-solving *per se*. Rather, its focus is on modeling the way in which fluid concepts allow analogies between different situations to be perceived in a natural and psychologically plausible manner. It is concerned with analogical *perception* (and, in particular, with *self*-perception), not analogical *reasoning* employed as a tool for solving problems, as in CBR. Furthermore, the emphasis in much CBR work is on systems that learn to solve problems in a progressively faster and more efficient manner, whereas in Metacat the notion of learning to perceive analogies with ever increasing efficiency and speed is irrelevant.

Metacat is actually closer to work on derivational analogy than to ordinary case-based approaches that store only a final problem solution. In derivational analogy, an entire trace of a problem-solving session is stored for future reference, not just the solution produced in the end, together with a series of annotations describing the conditions under which each step in the solution was taken [Carbonell, 1986; Veloso and Carbonell, 1993; Veloso, 1994]. In Metacat, the thematic information stored with an answer summarizes the important concepts and events that together contributed to the discovery of the answer, much like the temporal problem-solving trace of derivational analogy—although in Metacat's case, instead of storing the entire trace, an abstract description of the answer based on the information in the trace is stored.

In contrast to derivational analogy and CBR, however, Metacat (like Copycat) is deeply concerned with the nature of *concepts*. One of the prime objectives of this research is to explore how fluidly-adaptable concepts can give rise to understanding by enabling analogies between disparate situations to be perceived. Metacat's concepts, to be sure, come nowhere close to exhibiting the full power and fluidity of human concepts. Nevertheless, there is a sense in which they *are* genuinely meaningful entities—not just empty static symbols that get shunted around by the program. A concept node in the Slipnet—*successor*, for example—responds to the situation at hand in a continuous, context-dependent way, reflecting the degree of perceived relevance or presence of the idea of successorship in the Workspace at any given moment. A wide range of superficially dissimilar strings can in principle activate it—strings such as ***abc***, ***ijk***, ***pqrstu***, ***iijjkk***, ***mrrjjj***, ***mmrrrjjjj***, and ***aababcabcd***. Given the program's ability to flexibly recognize a wide range of instances of the same concept, some of them quite abstract, it is fair to say that Metacat'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 a more complete discussion of this point).

Work on Metacat is aimed at deepening Copycat's understanding of its answers by incorporating mechanisms for memory, reminding, and self-watching into the program. Many important ideas from case-based reasoning are relevant to this aim, such as the storing of past experiences as a repertoire of cases in memory, the activation of stored cases by similar situations, and the issue of analogical similarity of different situations. Unfortunately, case-based reasoning research concentrates on these issues at the expense of understanding the nature of concepts. Indeed, it seems likely that CBR's ultimate success—at least as a cognitive model—will be limited on account of its avoidance of this very difficult but critically important question. In contrast, Metacat can be seen as an attempt to broaden and enrich these ideas by focusing on them within a framework of fluid, context-sensitive concepts.

CHAPTER THREE

Generalizing the Representation of Rules

The broad-brush overview of the Metacat architecture presented in the previous chapter outlined several new architectural components and mechanisms not present in Copycat (*i.e.*, themes, the Episodic Memory, the Temporal Trace, and the Themespace). These components provide the basis for Metacat's self-watching ability, and together represent the model's central theoretical advance over the Copycat model. However, the development of the Metacat architecture has also involved extending and generalizing mechanisms that were present in the earlier model, such as the mechanisms for building mappings between strings, and for creating rules that describe differences between strings. This chapter examines these architectural refinements in detail. A thorough discussion of Metacat's new architectural components for self-watching, including the Themespace, the Temporal Trace, and the Episodic Memory, will be given in the next chapter. The focus of the present chapter is primarily on Metacat's generalized rule-building mechanisms, since it is here that restrictions inherent in the earlier model have been overcome to the greatest extent. However, several improvements to other previously-existing Copycat mechanisms will also be described.

3.1 Similarities and Differences Between Strings

In order to construct a rule that describes how the initial string changes into the modified string, Copycat must examine the ways in which the components of the strings correspond to each other, as well as how they differ. For example, in order to describe ***abc*** ⇒ ***abd*** as *Replace letter-category of rightmost letter by successor*, the ***c*** in ***abc*** must be seen as corresponding to the ***d*** in ***abd***. Said another way, ***c*** and ***d*** must be seen as *playing the same role* in their respective situations—in this case, the role of "rightmost letter in the string". Likewise, the two ***a***'s must correspond to each other, as must the two ***b***'s. Together, these correspondences effectively "align" the strings in a straightforward, one-to-one fashion, capturing the essential *similarity* shared by each letter of ***abc*** with its counterpart letter in ***abd***—namely, the identical positions occupied by corresponding letters within their respective strings.

In turn, this alignment highlights the *difference* between the strings, bringing out the ***c*** ⇒ ***d*** change clearly. This change can then be described in various ways using different levels of abstraction. For example, ***c*** can be described as either changing to its *successor* (viewing the ***c*** ⇒ ***d*** change abstractly) or as changing literally to the letter ***d*** (viewing the change in a more concrete way). The first interpretation results in the rule *Replace letter-category of rightmost letter by successor*, while the second results in *Replace letter-category of rightmost letter by 'd'.* Furthermore, the letter ***c*** itself can be described either abstractly as the "rightmost letter" of ***abc*** (as in the above two rules), or literally as the 'c' of ***abc***. Seeing things in the latter way results in the even more literal-minded rule *Replace 'c' by 'd'.* These different ways of describing ***abc*** ⇒ ***abd*** may give rise to different answers, depending on the problem. For instance, in the problem *"**abc** ⇒ **abd**; **ijk** ⇒ **?**"*, these three rules yield, respectively, the answers ***ijl***, ***ijd***, and ***ijk***.

In general, the bridges making up the mapping between the initial string and the modified string model the way in which the strings are perceived as being the *same*, while the rule models the way in which they are perceived as being *different.* These differences are perceived against the "background of similarity" provided by the mapping.

Another example that illustrates this "figure/ground" relationship is the problem *"**pq** ⇒ **qp**; **ijkl** ⇒ **?**"*. One way of perceiving ***pq*** ⇒ ***qp*** is to regard ***p*** and ***q*** as swapping positions. This interpretation, supported by the bridges ***p***–***p*** and ***q***–***q***, suggests swapping the positions of the left and right letters of ***ijkl***, yielding the answer ***ljki***. The "background of similarity" in this case is the idea of *letter-category invariance*—seeing the two ***p***'s and the two ***q***'s as corresponding to each other and thus implying that their *positions* change. Consequently, a rule based on this mapping would specify swapping the positions of the left and right letters of a string.

On the other hand, it is possible to see ***pq*** ⇒ ***qp*** differently—as ***p*** changing to its successor and ***q*** changing to its predecessor. This interpretation suggests changing the left letter of ***ijkl*** to its successor and the right letter to its predecessor, yielding ***jjkk***. Although people might not think of this answer at first, most would probably agree, in retrospect, that it makes sense. Here, the similarity between ***pq*** and ***qp*** rests on the idea of *position-invariance*—seeing the two *left* letters and the two *right* letters as corresponding to each other—supported by the bridges ***p***–***q*** and ***q***–***p***. A rule based on this mapping would specify changes to the *letter-categories* of the left and right letters of a string, rather than changes to their positions.

A third possibility, of course, is to regard ***pq*** as reversing direction *as a whole*, yielding the answer ***lkji***. Rather than focusing on the individual letters ***p*** and ***q***, this interpretation sees ***pq*** and ***qp*** as being *the same group*, implying that ***pq***'s *direction* changes. A mapping representing this idea of object-invariance at the group level would consist of a single bridge between the oppositely-directed groups ***pq*** and ***qp***, which would in turn give rise to a rule that specifies reversing the direction of a string as a whole.

3.2 Building Rules in Copycat

In Copycat, the ability to build arbitrary bridges between strings—and thus to characterize their similarities—is restricted to the initial string and target string only. Bridges between the initial string and modified string (which Mitchell calls "replacements") are only allowed between identically-positioned letters in each string, and no bonds or groups can be built in the modified string. Thus, the only allowable mappings on which to base a rule are simple one-to-one isomorphisms between strings of identical length. Furthermore, rules describing more than a single letter change are not possible, limiting the program's ability to characterize differences between strings. For example, all of the rules describing ***pq*** ⇒ ***qp*** mentioned above are impossible for Copycat to build, since the first rule requires building the bridges ***p***–***p*** and ***q***–***q***, the second involves changes to both letters of ***pq***, and the third requires chunking the letters of ***qp*** into a group. Although the program can handle problems involving ***abc*** ⇒ ***abd***, it cannot handle other seemingly-simple string changes such as ***abc*** ⇒ ***abcc***, ***abc*** ⇒ ***abcd***, or ***pq*** ⇒ ***ppqq***.

Once all replacements have been built between the initial string and the modified string (an essentially trivial task, given the restrictions imposed on the mapping), Copycat builds a rule by simply filling in a predefined template of the form:

Replace ______ *of* ______ *by* ______

The "slot-fillers" of a rule template are individual concepts from the Slipnet, such as *letter*, *Letter-category*, *rightmost*, or *successor*. The concepts making up a rule specify: (1) *which* object changes in a string; (2) which *aspect* of that object changes; and (3) *how* that aspect changes. For example, the rule *Replace letter-category of rightmost letter by successor* consists of: (1) the concepts *letter*, *String-Position*, and *rightmost*, specifying that the object to be changed is the string's rightmost letter; (2) the concept *Letter-Category*, specifying that the letter's letter-category changes (as opposed to, say, the letter's position); and (3) the concept *successor*, specifying that the letter-category changes to its successor (as opposed to, say, 'd' or some other letter). The important point is that, internally, rules are structured collections of Slipnet concepts. (Outwardly, rules are rendered in the form of short English phrases, but this is really just a surface-level "gloss" masking the underlying conceptual representation.)

3.3 Building Rules in Metacat

3.3.1 Generalized mappings between strings

In Metacat, the restrictions on bridges built between the initial string and the modified string have been relaxed. Any object of the initial string may map onto any object of the modified string, as long as the resulting bridge is supported by concept-mappings. Furthermore, bonds and groups can now be built inside the modified string. The same codelet-based processes used in Copycat to build structures in the initial string and target string (as well as between them) now extend to all three Workspace strings. Thus, Metacat's *Bond-scout*, *Group-scout*, and *Description-scout* codelets probabilistically choose from among objects in all three strings (according to the same criteria used in Copycat for just the initial string and the target string) when scouting for possible new structures to propose. In addition, Copycat's *Correspondence-scout* and *Replacement-finder* codelets have been superseded in Metacat by *Bridge-scout* codelets, which are capable of proposing either *vertical bridges* between objects in the initial string and target string, or *horizontal bridges* between objects in the initial string and modified string. Metacat's vertical bridges are exactly the same as Copycat's *correspondences*, whereas horizontal bridges are generalizations of Copycat's *replacements*. Although horizontal bridges differ slightly from vertical bridges in the types of concept-mappings that can support them (to be explained below), they

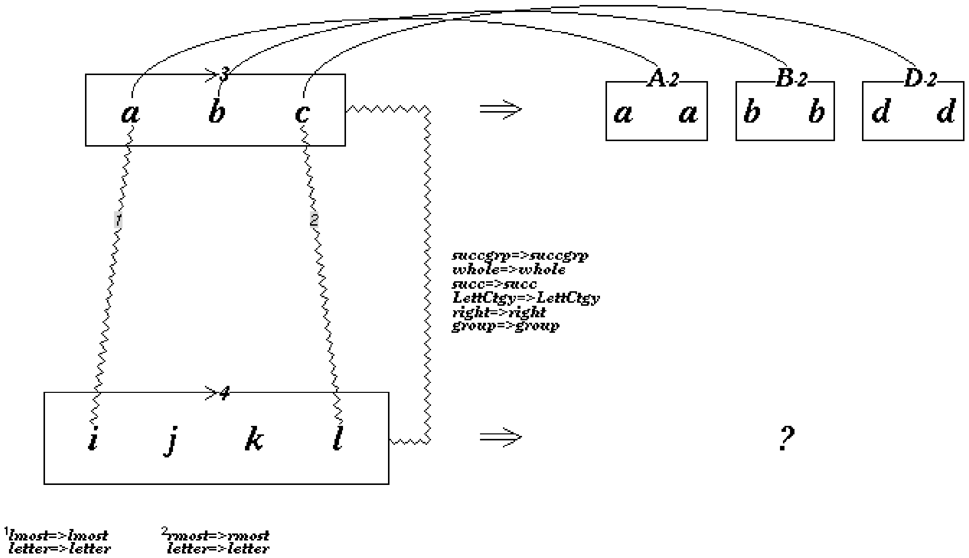


Figure 3.1: *Horizontal and vertical mappings created by Metacat for the problem "**abc** ⇒ **aabbdd**; **ijkl** ⇒ ?". Concept-mappings underlying the horizontal bridges are not shown.*

are fundamentally equivalent to vertical bridges. Both types of bridges link objects that play similar roles in a particular context.

An example illustrating Metacat's ability to build generalized string mappings is the problem *"**abc** ⇒ **aabbdd**; **ijkl** ⇒ ?"*. Figure 3.1 shows a possible state of the Workspace for this problem in which the letters of ***abc*** are mapped horizontally (according to string position) to the sameness groups ***aa***, ***bb***, and ***dd*** in the modified string. Likewise, vertical bridges map ***abc*** onto the target string ***ijkl***. Each bridge is supported by a set of concept-mappings, although only those associated with vertical bridges are displayed in Figure 3.1. In particular, the horizontal ***a***–***aa*** bridge is supported by the concept-mappings *leftmost* ⇒ *leftmost*, *letter* ⇒ *group*, *one* ⇒ *two* (since a letter of length one maps to a group of length two), and *a* ⇒ *a*; the ***b***–***bb*** bridge is supported by *middle* ⇒ *middle*, *letter* ⇒ *group*, *one* ⇒ *two*, and *b* ⇒ *b*; and ***c***–***dd*** is supported by *rightmost* ⇒ *rightmost*, *letter* ⇒ *group*, *one* ⇒ *two*, and *c* ⇒ *d*.

On the other hand, the vertical ***a***–***i*** and ***c***–***l*** bridges are supported by the concept-mappings *leftmost* ⇒ *leftmost*, *rightmost* ⇒ *rightmost*, and *letter* ⇒ *letter*, but not the slippages $a \Rightarrow i$ or $c \Rightarrow l$.

This example points out a subtle difference between horizontal and vertical bridges. Both types of bridges can be supported by *identity* concept-mappings involving length or letter-category, such as *three* ⇒ *three* or $a \Rightarrow a$, but *slippages* involving length or letter-category, such as *one* ⇒ *two* or $c \Rightarrow d$, are only possible for horizontal bridges. (Both horizontal and vertical bridges, however, can be supported by slippages based on concepts other than length or letter-category, such as *letter* ⇒ *group*.) Thus, the vertical ***a***–***i*** bridge lacks an $a \Rightarrow i$ slippage, the ***c***–***l*** bridge lacks a $c \Rightarrow l$ slippage, and the bridge mapping the length-three group ***abc*** as a whole to the length-four group ***ijkl*** lacks a *three* ⇒ *four* slippage.

The fundamental reason for this asymmetry is that horizontal bridge concept-mappings serve as the basis for characterizing *both similarities and differences* between strings, while vertical bridge concept-mappings serve only as the basis for characterizing *similarities* between strings. A rule must be "abstracted" from the concept-mappings underlying the horizontal bridges that explicitly describes how the initial string changes into the modified string, but no such rule needs to be created from the vertical bridges, since the notion of the initial string "changing into" the target string is not meaningful. Surface-level differences between letter-categories or lengths of objects are important when characterizing the changes that occur between strings, but they are not as important in the perception of similarity between strings. For example, in the problem *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"*, both ***abc*** and ***mrrjjj*** can be seen as successor groups, despite differences between the letter-categories and lengths of their components, whereas the change from ***abc*** to ***abd*** cannot be adequately described without taking into account the letter-category difference between ***c*** and ***d***. This point should become clearer when the rule abstraction process is described in greater detail.

As in Copycat, Metacat's Workspace objects have dynamically-varying, context-dependent levels of *importance* and *happiness* associated with them, which together determine an object's *salience*—its overall attractiveness to codelets. The importance of an object is a function of the object's descriptions (specifically, the number of currently-relevant descriptions, and the degree to which they are activated). The happiness of an object depends on the strengths of the bonds, groups, and bridges that together integrate the object into its surrounding context. Happiness reflects both how well the object contributes to the internal organization of its string, and how well it "fits into" the mappings constructed between strings.

However, unlike in Copycat, an object in Metacat's Workspace may be part of more than one mapping between strings. For example, in Figure 3.1, the ***a*** in ***abc*** corresponds "horizontally" to the ***aa*** group in ***aabbdd***, as well as "vertically" to the letter ***i*** in ***ijkl***. In general, the horizontal mapping between the initial string and modified string is independent of the vertical mapping between the initial string and target string. Objects that are part of both mappings (*i.e.*, objects in the initial string) therefore maintain separate happiness values for each mapping.[1] Thus, in the previous example, the ***b*** in ***abc*** has a lower *vertical* happiness value than either ***a*** or ***c***, since it remains unmapped to any object in ***ijkl***, but the *horizontal* happiness values of all three letters are about the same, since they all map to groups in ***aabbdd***. Since an object's salience depends on its happiness, objects with separate horizontal and vertical happiness values also have separate salience values for each mapping. This distinction is important mainly to *Bridge-scout* codelets, which tend to focus on the weaker mapping when looking for possible bridges to propose.

[1]This is also true for objects in the target string when Metacat "works backwards" from a given answer to an understanding of that answer. In this case, an additional horizontal mapping is created between the target string and the answer string. Metacat's ability to justify a given answer in this manner will be discussed in section 4.3 of Chapter 4.

3.3.2 From mappings to rules

To build a rule, codelets examine the concept-mappings underlying the horizontal bridges built between the initial string and modified string. They look for regularities among these concept-mappings—particularly among the *slippages*—and then build a high-level description of the mapping based on the patterns they have noticed. Unlike in Copycat, there are no fixed rule templates to be filled in which determine *a priori* the internal structure of rules. Instead, rules in Metacat are constructed according to a set of basic guidelines (outlined below) that allow rules of arbitrary length and complexity to be created, at least in principle.

Changes to objects in a string can be described either "intrinsically" or "extrinsically". An *intrinsic change* describes a single object that changes in some way—for example, the letter ***c*** changing to its successor in ***abc*** ⇒ ***abd***. An *extrinsic change*, on the other hand, describes several objects that change in some way relative to each other (by exchanging some attribute among themselves). An example would be two letters that swap their positions in a string, such as ***p*** and ***q*** in ***pq*** ⇒ ***qp***.

In general, the guidelines for constructing rules are as follows:

1. An intrinsic change can involve any aspect of an object (not just the object's letter-category).[2] In ***abc*** ⇒ ***abcc***, for example, ***c***'s *length* changes from one to two, and its *object-type* changes from a letter to a group (assuming that the letter ***c*** in ***abc*** is seen as corresponding to the group ***cc*** in ***abcc***).

2. An extrinsic change can involve two or more objects. For example, ***eqe*** ⇒ ***qeq*** can be described extrinsically as a letter-category swap (involving the letter-categories *e* and *q*) among all three letters of the string ***eqe***.

[2]The one exception to this is that a change to an object's string position cannot be described intrinsically—although it can be described *extrinsically* in conjunction with other objects whose positions also change.

3. For a particular set of objects, an extrinsic change can specify exchanging any number of *attributes* of the objects. For example, in ***eeqee*** ⇒ ***qeeq***, the objects ***ee***, ***q***, and ***ee*** in ***eeqee*** can be described as swapping their *lengths* as well as their *letter-categories* (assuming that these objects are seen as corresponding, respectively, to ***q***, ***ee***, and ***q*** in ***qeeq***).

4. Both intrinsic and extrinsic changes can designate *all of the components* of a particular object as changing—rather than the object itself. For example, ***abc*** ⇒ ***xxx*** can be described by a single intrinsic change specifying that all objects in the string ***abc*** change to ***x***.

5. A rule can specify any number of intrinsic or extrinsic changes to a string (in any combination).

6. Finally, a special type of rule called a *verbatim rule* can describe an entire string as changing literally to a particular sequence of letters. For example, ***abc*** ⇒ ***pxgk*** can be described in this way. Verbatim rules do not specify any other types of changes.

Like rules in Copycat, Metacat's rules are represented internally as structured collections of Slipnet concepts, but are expressed outwardly in the form of short English phrases. The English-like appearance of rules, however, is somewhat deceptive. There is nothing like an embedded "natural language module" for rules in Metacat. Instead, the constraints placed on the structure of rules ensure that the set of concepts making up a rule can be transcribed into passable English in a relatively straightforward way, avoiding the need for sophisticated linguistic processing. Unfortunately, Metacat's "English" falls somewhat short of native-level mastery, occasionally resulting in a vague or awkward-sounding rule. However, this is not really important from a theoretical standpoint, since the purpose of rules is to represent relationships between strings in abstract *conceptual* terms, rather than in a more concrete linguistic form.

In some ways, this emphasis on the conceptual level is reminiscent of Schank's conceptual-dependency approach to mental representations, in which a small set of abstract primitives is used to represent a wide variety of concrete, real-world situations [Schank, 1975; Schank and Abelson, 1977]. In Schank's theory, the level of representation that really counts is that of the underlying conceptual primitives. Likewise, the same is true for Metacat's rules. The English appearance of rules is really just a surface-level *façade*. However, unlike the primitives of Schank's theory, the conceptual primitives out of which Metacat's rules are built (*i.e.*, Slipnet concepts) are not empty, static entities; rather, they are *active* representational structures that respond in dynamic and context-sensitive ways to the pattern of perceptual activity occurring in the Workspace.

3.3.3 A sampler of Metacat rules

This section presents a representative sampling of the types of rules that Metacat is able to build. These examples illustrate more clearly the various guidelines for constructing rules given in the previous section. Each example shows a pair of strings along with one or more possible rules describing how the first string changes into the second string. Since the creation of a rule depends on the particular mapping created between the strings, the bridges between corresponding string objects are indicated as well. In general, mappings may be described in more than one way, depending on the level of abstraction used to refer to the string objects. For this reason, many of the examples below list several possible rules for a single pair of strings (although not every possible rule is necessarily shown).

The following rules describe intrinsic changes involving various attributes of string objects, such as letter-category or length. The examples in this group illustrate the first point appearing in the list of rule-creation guidelines given earlier.

- $\boldsymbol{abc} \Rightarrow \boldsymbol{abd}$ (assuming a mapping with bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{b}$–$\boldsymbol{b}$, and $\boldsymbol{c}$–$\boldsymbol{d}$)
    - *Change letter-category of rightmost letter to successor*
    - *Change letter-category of rightmost letter to 'd'*
    - *Change letter-category of letter 'c' to 'd'*
- $\boldsymbol{abc} \Rightarrow \boldsymbol{abcc}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{b}$–$\boldsymbol{b}$, and $\boldsymbol{c}$–$\boldsymbol{cc}$)
    - *Increase length of rightmost letter by one*
    - *Increase length of letter 'c' by one*
- $\boldsymbol{abc} \Rightarrow \boldsymbol{abccc}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{b}$–$\boldsymbol{b}$, and $\boldsymbol{c}$–$\boldsymbol{ccc}$)
    - *Change rightmost letter to a group of length three*
- $\boldsymbol{aaa} \Rightarrow \boldsymbol{a}$ (assuming a bridge $\boldsymbol{aaa}$–$\boldsymbol{a}$)
    - *Change whole group to a letter*
- $\boldsymbol{a} \Rightarrow \boldsymbol{z}$ (assuming a bridge $\boldsymbol{a}$–$\boldsymbol{z}$)
    - *Change alphabetic-position of single letter to opposite*
- $\boldsymbol{abc} \Rightarrow \boldsymbol{cba}$ (assuming a bridge $\boldsymbol{abc}$–$\boldsymbol{cba}$)
    - *Reverse direction of whole group*
- $\boldsymbol{axc} \Rightarrow \boldsymbol{cxa}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{x}$–$\boldsymbol{x}$, and $\boldsymbol{c}$–$\boldsymbol{c}$)
    - *Reverse direction of string*
- $\boldsymbol{abc} \Rightarrow \boldsymbol{abcd}$ (assuming a bridge $\boldsymbol{abc}$–$\boldsymbol{abcd}$)
    - *Increase length of whole group by one*
    - *Change length of whole group to four*

- ***abcd*** ⇒ ***abc*** (assuming a bridge ***abcd***–***abc***)
  - *Decrease length of whole group by one*
- ***aabbcc*** ⇒ ***aabbccdd*** (assuming a bridge ***aabbcc***–***aabbccdd***)
  - *Increase length of whole group by one*

Rules describing extrinsic changes, in which one or more object attributes are exchanged among several different string objects, are shown in the next set of examples. These examples illustrate the second and third points appearing in the list of guidelines given earlier.

- ***abc*** ⇒ ***cba*** (assuming bridges ***a***–***a***, ***b***–***b***, and ***c***–***c***)
  - *Swap positions of leftmost letter and rightmost letter*
  - *Swap positions of letter 'a' and letter 'c'*
- ***abc*** ⇒ ***cba*** (assuming bridges ***a***–***c***, ***b***–***b***, and ***c***–***a***)
  - *Swap letter-categories of leftmost letter and rightmost letter*
- ***aabccc*** ⇒ ***aaabcc*** (assuming bridges ***aa***–***aaa***, ***b***–***b***, and ***ccc***–***cc***)
  - *Swap lengths of leftmost group and rightmost group*
- ***aabccc*** ⇒ ***aacbbb*** (assuming bridges ***aa***–***aa***, ***b***–***c***, and ***ccc***–***bbb***)
  - *Swap letter-categories of middle group and rightmost group*
- ***aabaa*** ⇒ ***bbabb*** (assuming bridges ***aa***–***bb***, ***b***–***a***, and ***aa***–***bb***)
  - *Swap letter-categories of leftmost group, middle letter, and rightmost group*
  - *Swap letter-categories of leftmost group, letter 'b', and rightmost group*

- ***azx*** $\Rightarrow$ ***zax*** (assuming bridges ***a***–***a***, ***z***–***z***, and ***x***–***x***)
  - *Swap positions of letter 'a' and letter 'z'*
  - *Swap positions of alphabetic-first letter and alphabetic-last letter*
- ***eqe*** $\Rightarrow$ ***qeq*** (assuming bridges ***e***–***q***, ***q***–***e***, and ***e***–***q***)
  - *Swap letter-categories of leftmost letter, middle letter, and rightmost letter*
- ***eeqee*** $\Rightarrow$ ***qeeq*** (assuming bridges ***ee***–***q***, ***q***–***ee***, and ***ee***–***q***)
  - *Swap letter-categories and lengths of leftmost group, middle group, and rightmost group*

The next set of examples shows how both intrinsic and extrinsic changes can refer to all of the *components* of a particular object. This is a powerful abstraction capability, which allows systematic changes across a string to be captured in a natural and succinct manner. The real power behind this way of describing string changes becomes apparent when rules are translated and applied to new situations (*i.e.*, to new strings).

For example, in the problem *"**eqe** $\Rightarrow$ **qeq**; **axaxa** $\Rightarrow$ **?**"*, viewing the ***eqe*** $\Rightarrow$ ***qeq*** change as a letter-category swap involving all of the letters of ***eqe*** implies swapping all of the letters of ***axaxa***, yielding the answer ***xaxax***, whereas viewing the swap as explicitly involving the leftmost, middle, and rightmost letters of ***eqe*** implies swapping only the ***a***'s in ***axaxa***, which leaves the string unchanged.

The examples below serve to illustrate the fourth point in the list of guidelines given earlier.

- ***abc*** $\Rightarrow$ ***bcd*** (assuming bridges ***a***–***b***, ***b***–***c***, and ***c***–***d***)
  - *Change letter-categories of all objects in whole group to successor*

- $\boldsymbol{abc} \Rightarrow \boldsymbol{aabbcc}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{aa}$, $\boldsymbol{b}$–$\boldsymbol{bb}$, and $\boldsymbol{c}$–$\boldsymbol{cc}$)
  - *Increase lengths of all objects in whole group by one*
  - *Change all objects in whole group to groups of length two*
- $\boldsymbol{aaabbbccc} \Rightarrow \boldsymbol{aabbcc}$ (assuming bridges $\boldsymbol{aaa}$–$\boldsymbol{aa}$, $\boldsymbol{bbb}$–$\boldsymbol{bb}$, and $\boldsymbol{ccc}$–$\boldsymbol{cc}$)
  - *Decrease lengths of all objects in whole group by one*
  - *Change lengths of all objects in whole group to two*
- $\boldsymbol{abc} \Rightarrow \boldsymbol{aaa}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{b}$–$\boldsymbol{a}$, and $\boldsymbol{c}$–$\boldsymbol{a}$)
  - *Change letter-categories of all objects in whole group to 'a'*
- $\boldsymbol{abbccc} \Rightarrow \boldsymbol{abc}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{a}$, $\boldsymbol{bb}$–$\boldsymbol{b}$, and $\boldsymbol{ccc}$–$\boldsymbol{c}$)
  - *Change all objects in whole group to letters*
- $\boldsymbol{abbccc} \Rightarrow \boldsymbol{aaabbbccc}$ (assuming bridges $\boldsymbol{a}$–$\boldsymbol{aaa}$, $\boldsymbol{bb}$–$\boldsymbol{bbb}$, and $\boldsymbol{ccc}$–$\boldsymbol{ccc}$)
  - *Change lengths of all objects in whole group to three*
- $\boldsymbol{aabbcc} \Rightarrow \boldsymbol{xxx}$ (assuming bridges $\boldsymbol{aa}$–$\boldsymbol{x}$, $\boldsymbol{bb}$–$\boldsymbol{x}$, and $\boldsymbol{cc}$–$\boldsymbol{x}$)
  - *Change all objects in whole group to the letter 'x'*
- $\boldsymbol{eqe} \Rightarrow \boldsymbol{qeq}$ (assuming bridges $\boldsymbol{e}$–$\boldsymbol{q}$, $\boldsymbol{q}$–$\boldsymbol{e}$, and $\boldsymbol{e}$–$\boldsymbol{q}$)
  - *Swap letter-categories of all objects in string*
- $\boldsymbol{eeqee} \Rightarrow \boldsymbol{qeeq}$ (assuming bridges $\boldsymbol{ee}$–$\boldsymbol{q}$, $\boldsymbol{q}$–$\boldsymbol{ee}$, and $\boldsymbol{ee}$–$\boldsymbol{q}$)
  - *Swap letter-categories and lengths of all objects in string*

- ***abbbc*** ⇒ ***aaabccc*** (assuming bridges ***a***–***aaa***, ***bbb***–***b***, and ***c***–***ccc***)
  - *Swap lengths of all objects in whole group*

Several different changes to a string (both intrinsic and extrinsic) can be described by a single rule. The following examples involve strings that change in multiple ways. The rules in this series of examples illustrate the fifth point in the list of guidelines given earlier.

- ***abc*** ⇒ ***cba*** (assuming bridges ***a***–***c***, ***b***–***b***, and ***c***–***a***)
  - *Change letter-category of leftmost letter to 'c'*
    Change letter-category of rightmost letter to 'a'

- ***abc*** ⇒ ***abdd*** (assuming bridges ***a***–***a***, ***b***–***b***, and ***d***–***dd***)
  - *Change letter-category of rightmost letter to successor*
    Increase length of rightmost letter by one
  - *Change letter-category of rightmost letter to 'd'*
    Increase length of rightmost letter by one
  - *Change letter-category of rightmost letter to successor*
    Change rightmost letter to a group of length two
  - *Change rightmost letter to a 'd' group of length two*
  - *Change letter 'c' to a 'd' group of length two*

- ***abc*** ⇒ ***ddba*** (assuming bridges ***a***–***a***, ***b***–***b***, and ***c***–***dd***)
  - *Change letter-category of rightmost letter to successor*
    Increase length of rightmost letter by one
    Reverse direction of whole group
  - *Change rightmost letter to a 'd' group of length two*
    Reverse direction of whole group

- ***abc*** $\Rightarrow$ ***bbccdd*** (assuming bridges ***a***–***bb***, ***b***–***cc***, and ***c***–***dd***)
  - *Change letter-categories of all objects in whole group to successor*
    Increase lengths of all objects in whole group by one

- ***abc*** $\Rightarrow$ ***cbba*** (assuming bridges ***a***–***a***, ***b***–***bb***, and ***c***–***c***)
  - *Increase length of middle letter by one*
    Swap positions of leftmost letter and rightmost letter

  - *Increase length of middle letter by one*
    Reverse direction of whole group

- ***abc*** $\Rightarrow$ ***ccbbaa*** (assuming bridges ***a***–***aa***, ***b***–***bb***, and ***c***–***cc***)
  - *Reverse direction of whole group*
    Increase lengths of all objects in whole group by one

- ***eqe*** $\Rightarrow$ ***qqeeqq*** (assuming bridges ***e***–***qq***, ***q***–***ee***, and ***e***–***qq***)
  - *Increase lengths of all objects in string by one*
    Swap letter-categories of all objects in string

Finally, a *verbatim rule* describes changing one string into another in the most literal-minded way possible, essentially ignoring any relationships between corresponding letters or groups in the strings. Of course, if no such relationships exist to begin with, then the only way to describe the situation is with a verbatim rule. Two examples of such rules are shown below, rounding out the list of rule-creation guidelines given earlier. (In fact, all of the string changes given in the preceding examples could have been described by verbatim rules as well.)

- ***abc*** $\Rightarrow$ ***abd***
  - *Change string to "abd"*

- ***abc*** $\Rightarrow$ ***mrrjjj***
  - *Change string to "mrrjjj"*

3.3.4 The internal structure of rules in detail

To more clearly appreciate the flexibility—and limitations—of Metacat's rule representation, it is necessary to examine the internal structure of rules more precisely. Essentially, a rule consists of an arbitrarily-long list of *rule clauses*, of which there are three possible types:

- An *intrinsic clause* refers to a single object in a string, and specifies an arbitrary number of changes either to the object itself or to its components.
- An *extrinsic clause* refers to a *set* of objects in a string, and specifies one or more attributes that are exchanged among the objects.
- A *verbatim clause* does not refer to any objects in a string. It simply specifies a new sequence of letters to which the string changes.

In fact, a rule may contain no clauses at all, in which case it is the "identity rule" *Don't change anything.* Verbatim rules consist of exactly one verbatim clause. All other rules consist of combinations of intrinsic and extrinsic clauses.

The detailed structure of rules can be described in terms of a simple grammar, much like those used to specify the formal syntax of programming languages. This "rule grammar" is shown in Figure 3.2. A rule is essentially a nested list structure whose atomic elements are either Slipnet nodes, or special "tag" symbols that serve as markers for various intermediate-level structures. The identity rule is represented by the empty list. In the figure, Slipnet nodes are shown in slanted type (*e.g.*, *successor*) and tag symbols are shown in sans serif type (*e.g.*, extrinsic).

$$\langle \textit{rule} \rangle \longrightarrow (\{\langle \textit{intrinsic-clause} \rangle \mid \langle \textit{extrinsic-clause} \rangle\} \ldots) \mid (\langle \textit{verbatim-clause} \rangle) \mid (\,)$$

$$\langle \textit{intrinsic-clause} \rangle \longrightarrow (\textsf{intrinsic}\ (\langle \textit{object-description} \rangle)\ (\langle \textit{change-description} \rangle \ldots))$$

$$\langle \textit{extrinsic-clause} \rangle \longrightarrow (\textsf{extrinsic}\ (\langle \textit{object-description} \rangle \ldots)\ (\langle \textit{object-attribute} \rangle \ldots))$$

$$\langle \textit{verbatim-clause} \rangle \longrightarrow (\textsf{verbatim}\ (\langle \textit{platonic-letter} \rangle \ldots))$$

$$\langle \textit{object-description} \rangle \longrightarrow (\langle \textit{object-type} \rangle\ \langle \textit{object-attribute} \rangle\ \langle \textit{object-descriptor} \rangle) \mid (\textsf{string}\ \textit{String-Position}\ \textit{whole})$$

$$\langle \textit{object-type} \rangle \longrightarrow \textit{letter} \mid \textit{group}$$

$$\langle \textit{object-attribute} \rangle \longrightarrow \{\textit{any Slipnet category node}\}$$

$$\langle \textit{object-descriptor} \rangle \longrightarrow \{\textit{any Slipnet descriptor node}\}$$

$$\langle \textit{change-description} \rangle \longrightarrow (\langle \textit{referent} \rangle\ \langle \textit{object-attribute} \rangle\ \langle \textit{change-descriptor} \rangle)$$

$$\langle \textit{referent} \rangle \longrightarrow \textsf{object} \mid \textsf{components}$$

$$\langle \textit{change-descriptor} \rangle \longrightarrow \langle \textit{platonic-relation} \rangle \mid \langle \textit{object-descriptor} \rangle$$

$$\langle \textit{platonic-relation} \rangle \longrightarrow \textit{successor} \mid \textit{predecessor} \mid \textit{opposite}$$

$$\langle \textit{platonic-letter} \rangle \longrightarrow \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \ldots \mid z$$

Figure 3.2: *A grammar that describes the precise structure of Metacat's rules.*

The symbols intrinsic, extrinsic, and verbatim distinguish different types of rule clauses. Verbatim clauses—the simplest—contain just a list of platonic Slipnet letters representing a literal transformation to a new string consisting of exactly those letters. The rule *Change string to "abd"*, for example, consists of the single verbatim clause (verbatim (a b d)).

In contrast, both intrinsic and extrinsic clauses refer to specific objects in a string via *object-descriptions*. An object-description is a trio of Slipnet nodes that uniquely identifies a particular object in a string. The first node indicates whether the object is a letter or a group; the second and third nodes specify some attribute of the object that distinguishes it from other objects in its string. For example, an object-description consisting of the nodes *letter*, *String-Position*, and *rightmost* would refer to the letter ***c*** in the string ***abc***, or to the rightmost letter ***k*** in ***iijjkk***. An object-description consisting of the nodes *group*, *Length*, and *three* would refer to the group ***jjj*** in ***mrrjjj***, or to the group ***bbb*** in ***abbbc***. An object-description consisting of *letter*, *Letter-Category*, and b would refer to the letter ***b*** in ***abcd***.

In general, any Slipnet category node—in conjunction with any descriptor node associated with the category—may be used to designate an object. It is also possible for an object-description to refer to the string itself, rather than to a specific letter or group, even though there is no "string" concept in the Slipnet. In this case, the special symbol string is used as the object type, along with the attribute *String-Position* and the descriptor *whole*. Such an object-description, for example, could be used to refer to the string ***pxq***, which cannot otherwise be described as a unit (unlike the string ***abc***, which can be described as a group encompassing the whole string).

An extrinsic clause contains a list of object-descriptions specifying a set of objects in a string, as well as a list of the object-attributes that get exchanged among the designated objects (such as their string positions or letter-categories). Since any number of objects may "participate" in an exchange, any number of object-descriptions may appear in an extrinsic clause. For example, the rule *Swap letter-categories and lengths of leftmost group, middle letter, and rightmost group*, which could describe the string change ***eeqee*** ⇒ ***qeeq***, consists of an extrinsic clause containing three object-descriptions:

- (*group String-Position leftmost*)
- (*letter String-Position middle*)
- (*group String-Position rightmost*)

and two object-attributes: *Letter-Category* and *Length*. As a special case, if an extrinsic clause contains only *one* object-description, then the designated object's *components* are involved in the exchange, rather than the object itself. For example, ***eeqee*** ⇒ ***qeeq*** could be described more generally as *Swap letter-categories and lengths of all objects in string* (assuming that ***ee*** groups exist in ***eeqee***). The extrinsic clause for this rule would contain just one object description:

- (string *String-Position whole*)

and the same two object-attributes as before: *Letter-Category* and *Length*, indicating that the string's component objects ***ee***, ***q***, and ***ee*** exchange their letter-categories and lengths.

In contrast to an extrinsic clause, an intrinsic clause refers to just one object in a string, so only one object-description is needed. However, any number of changes to this object may be specified. Consequently, any number of *change-descriptions* are permitted in an intrinsic clause. Each change-description describes a particular change made either to the object itself, or to all of the object's components (if the object is not a letter). A change-description consists of three parts: a tag symbol that indicates whether the change applies to the object or to the object's components, the particular object-attribute being changed (such as *Letter-Category* or *Length*), and a descriptor that specifies *how* the attribute changes. This descriptor may be either an abstract relationship (*i.e.*, *successor*, *predecessor*, or *opposite*), or a literal descriptor (such as *d* or *three*). For example, the change-description

(object *Length successor*)

says to increase the length of an object by one. This change-description, in conjunction with the object-description (*group String-Position whole*), would transform the string ***abc*** to ***abcd*** (assuming that ***abc*** is seen as a string-spanning successor group). On the other hand, if the change-description's referent symbol were components instead of object, then the length of each *letter* in ***abc*** would be increased by one, resulting in the string ***aabbcc***. Alternatively, both of these string changes could be described in a literal way by using an explicit length descriptor in place of *successor* (*i.e.*, *four* in the case of ***abc*** ⇒ ***abcd***, or *two* in the case of ***abc*** ⇒ ***aabbcc***). Sometimes, using a literal change-descriptor is the only way to describe a change, as in the case of ***abc*** ⇒ ***abcde*** or ***abc*** ⇒ ***aaabbbccc*** (since the concept of "double-successorship" is unknown to Metacat). Finally, for any particular object-attribute, only certain platonic relations can be used to describe a change at an abstract level. For example, changing a group's length can be described abstractly by the relations *successor* and *predecessor*, but changing its direction can only be described by the relation *opposite*, since the idea of successorship or predecessorship makes no sense for a direction. These examples are summarized in Figure 3.3.

In general, any number of intrinsic or extrinsic clauses are permitted in a rule, so any number of independent changes—involving any number of objects in a string—can be expressed. In practice, however, most rules describe no more than two or three changes, simply because the more complicated it becomes to describe the transformation of one string into another, the more difficult it becomes for Metacat to discover any underlying similarity between the strings in the first place, on which it

String change	Object-description	Change-description
abc ⇒ ***abcd***	(*group String-Position whole*)	(object *Length successor*)
abc ⇒ ***abcd***	(*group String-Position whole*)	(object *Length four*)
abc ⇒ ***aabbcc***	(*group String-Position whole*)	(components *Length successor*)
abc ⇒ ***aabbcc***	(*group String-Position whole*)	(components *Length two*)
abc ⇒ ***abcde***	(*group String-Position whole*)	(object *Length five*)
abc ⇒ ***aaabbbccc***	(*group String-Position whole*)	(components *Length three*)
abc ⇒ ***cba***	(*group String-Position whole*)	(object *Direction opposite*)

Figure 3.3: *Examples of intrinsic change-descriptions applied to the string* ***abc*** *(seen as a string-spanning successor group).*

could then base a rule. For example, the string change ***abc*** ⇒ ***ddaxxx*** could theoretically be described as reversing the direction of ***abc*** while simultaneously changing (1) the letter-category and length of the rightmost letter ***c*** to its successor; (2) the letter-category of the middle letter ***b*** to its predecessor; and (3) the letter-category and length of the leftmost letter ***a*** to *x* and *three*, respectively. Such a contrived rule, while indeed expressible as a set of intrinsic-clauses, is unlikely to ever be built, since it relies rather arbitrarily on seeing ***a*** as corresponding to ***xxx*** and ***c*** as corresponding to ***dd*** (much more likely would be the verbatim rule *Change string to "ddaxxx"*). The point is that the *expressive* power of Metacat's rule formalism is substantially greater than Metacat's power to *discover* rules. In other words, Metacat's ability to describe differences between strings is limited more by the constraints that arise out of its mechanisms for perceiving similarity between strings than by restrictions imposed by the grammatical formalism used to represent rules.

We round out the present section by exhibiting the complete internal structure of several of the rules listed in the "Metacat rule sampler" given earlier (see Figure 3.4). The English language "façade" of each rule appears in a box immediately above the rule's actual representation in terms of Slipnet concepts, along with a pair of strings that could serve as the basis for building the rule.

Change alphabetic-position of single letter to opposite	$\boldsymbol{a \Rightarrow z}$

```
((intrinsic ((letter String-Position single))
            ((object Alphabetic-Position opposite))))
```

Reverse direction of whole group *Increase lengths of all objects in whole group by one*	$\boldsymbol{abc \Rightarrow ccbbaa}$

```
((intrinsic ((group String-Position whole))
            ((object Direction opposite)
             (components Length successor)
             (components Object-Category group))))
```

Change rightmost letter to a 'd' group of length two *Reverse direction of whole group*	$\boldsymbol{abc \Rightarrow ddba}$

```
((intrinsic ((letter String-Position rightmost))
            ((object Letter-Category d)
             (object Object-Category group)
             (object Length two)))
 (intrinsic ((group String-Position whole))
            ((object Direction opposite))))
```

Swap lengths of leftmost group and rightmost group	$\boldsymbol{aabccc \Rightarrow aaabcc}$

```
((extrinsic ((group String-Position leftmost)
             (group String-Position rightmost))
            (Length)))
```

Increase lengths of all objects in string by one *Swap letter-categories of all objects in string*	$\boldsymbol{eqe \Rightarrow qqeeqq}$

```
((intrinsic ((string String-Position whole))
            ((components Length successor)
             (components Object-Category group)))
 (extrinsic ((string String-Position whole))
            (Letter-Category)))
```

Figure 3.4: *The complete internal structure of several rules from section 3.3.3.*

One last point is worth mentioning. In general, if a string change involves one or more letters that change their length, then the corresponding rule structure will include change-descriptions stipulating that *Object-Category* changes to *group*—since increasing a letter's length necessarily turns it into a group—in addition to the appropriate *Length* change-descriptions. However, in order to avoid redundancy, the English rendition of such a rule does not usually mention these "extra" change-descriptions explicitly. The second and fifth rule examples in Figure 3.4 illustrate this point (although the third rule, which is more literal than the others, does mention the letter-to-group change explicitly). In any case, regardless of a rule's outward English appearance, all of the information that uniquely characterizes the rule is present in its underlying conceptual structure, which is the only representational level that really matters.

3.3.5 Measures of rule quality

In Copycat, the strength of a rule is a function of (1) the conceptual depths of its descriptors, and (2) whether or not these descriptors play a role in the current mapping between the initial string and target string (more specifically, between the changed letter in the initial string and its corresponding object in the target string). For example, to describe ***abc*** ⇒ ***abd***, Copycat may build the rule *Replace letter-category of rightmost letter by successor* or the rule *Replace 'c' by 'd'*. The former rule involves the concepts *rightmost* and *successor*, which have greater conceptual depth than the concepts *c* or *d*, so this rule is in general the stronger one—unless the concept of *c* (rather than *rightmost*) happens to apply to ***c***'s corresponding object in the target string. This would be the case in the problem *"**abc** ⇒ **abd**; **xccx** ⇒ **?**"* if the ***c*** in ***abc*** were mapped to the middle ***cc*** group in ***xccx*** (based on the concept-mapping *c* ⇒ *c*), rather than to the rightmost letter ***x*** (based on the concept-mapping *rightmost* ⇒ *rightmost*).

In Metacat, the calculation of rule strength is less straightforward, because the internal structure of rules may be arbitrarily complicated. The quality of a rule in Metacat depends not only on the conceptual depths of the rule's constituent concepts, but also on the internal "coherence" of these concepts. More precisely, the quality of a rule is a function of three independent measures:

- The *uniformity* of a rule, which reflects to what extent different rule clauses describe objects (or changes to those objects) in the same way.
- The *abstractness* of a rule, which reflects the depths of the concepts used to describe objects (or changes to those objects).
- The *succinctness* of a rule, which reflects the number of rule clauses used to describe objects (or changes to those objects).

All of these measures, to be discussed below, depend only on the internal structure of a rule. In the current version of Metacat, the quality of a rule is not sensitive to the perceptual context in which the rule is used (unlike in Copycat, as described above), although ideally it should be. The situation in Metacat is complicated by the fact that a single rule may specify any number of objects as changing in the initial string. Some of these objects may be described in the same way as their corresponding target string objects (as in the case of a *leftmost* object mapping to a *leftmost* object), while others may be described differently (as in the case of a *rightmost* object mapping to a *'c'* object). All of these potentially conflicting considerations would need to be taken into account in order to judge the quality of a rule with respect to a particular perceptual context. In other words, the quality of a rule should be a function both of the rule's internal structure and of the string mappings that exist in the Workspace. This problem should eventually be remedied. Nevertheless, the present method of calculating rule quality suffices to allow rules to be qualitatively ranked, relative to one another, in a plausible fashion.

Uniformity

As explained in section 3.3.4, rule clauses refer to objects in a string in terms of attributes such as string position (*e.g.*, the *rightmost* letter) or letter-category (*e.g.*, the *'c'* letter). When several objects change in a string, there should be pressure to refer to these objects in a uniform way, using the same object-attribute. Likewise, there should be pressure to describe the *changes* in a uniform way—either all in terms of abstract relationships (such as *successor*), or all in terms of literal descriptors (such as d). The uniformity of a rule is thus a function of:

1. *The uniformity of the intrinsic-clause object-description attributes.* For example, the string change ***abc*** ⇒ ***xxx*** could be described by each of the three rules shown below:

   (1) *Change letter-category of leftmost letter to 'x'*
   Change letter-category of middle letter to 'x'
   Change letter-category of rightmost letter to 'x'

   (2) *Change letter-category of letter 'a' to 'x'*
   Change letter-category of letter 'b' to 'x'
   Change letter-category of letter 'c' to 'x'

   (3) *Change letter-category of alphabetic-first letter to 'x'*
   Change letter-category of letter 'b' to 'x'
   Change letter-category of rightmost letter to 'x'

   Rule (1) and Rule (2) are equally uniform, since both identify all of the changed objects either in terms of their string positions or their letter-categories. In contrast, Rule (3) describes the objects using a heterogeneous mixture of three different object-attributes, so it is less uniform than the other two rules.

2. *The uniformity of the extrinsic-clause object-description attributes.* For example, the string change ***abc*** ⇒ ***cba*** could be described by each of the three rules shown below:

(1) *Swap positions of leftmost letter and rightmost letter*

(2) *Swap positions of letter 'a' and letter 'c'*

(3) *Swap positions of letter 'a' and rightmost letter*

As in the previous example, the first two rules are equally uniform, and both are more uniform than the third rule.

3. *The uniformity of the intrinsic-clause change-descriptors.* For example, the string change ***abc*** ⇒ ***bad*** could be described by each of the three rules shown below:

   (1) *Change letter-category of leftmost letter to successor*
   Change letter-category of middle letter to predecessor
   Change letter-category of rightmost letter to successor

   (2) *Change letter-category of leftmost letter to 'b'*
   Change letter-category of middle letter to 'a'
   Change letter-category of rightmost letter to 'd'

   (3) *Change letter-category of leftmost letter to successor*
   Change letter-category of middle letter to 'a'
   Change letter-category of rightmost letter to 'd'

   In Rule (1), all of the changes are described in terms of abstract successor or predecessor relationships; in Rule (2) all changes are described literally. These two rules are thus equally uniform (although they are not equally abstract). However, both are more uniform than Rule (3), which describes the changes using a mixture of abstract and concrete descriptors.

4. *The uniformity of the clause types used to describe changes.* For example, the string change ***eqe*** ⇒ ***qeq*** could be described by each of the three rules shown below:

(1) *Change letter-category of leftmost letter to 'q'*
Change letter-category of middle letter to 'e'
Change letter-category of rightmost letter to 'q'

(2) *Swap letter-categories of leftmost letter, middle letter, and rightmost letter*

(3) *Change letter-category of leftmost letter to 'q'*
Swap letter-categories of middle letter and rightmost letter

Rule (1) describes all of the changes intrinsically, while Rule (2) describes the changes extrinsically. Both of these rules are more uniform than Rule (3), which describes the changes using a mixture of intrinsic and extrinsic clauses.

Abstractness

The quality of a rule also depends on the average conceptual depth of its descriptors. In general, abstract descriptions of string changes are preferred over literal descriptions. The abstractness of a rule is thus a function of:

1. *The average conceptual depth of object-description attributes.* For example, both of the rules shown below could describe the string change ***abc*** $\Rightarrow$ ***xxx***, but the first rule is more abstract than the second:

   (1) *Change letter-category of leftmost letter to 'x'*
   Change letter-category of middle letter to 'x'
   Change letter-category of rightmost letter to 'x'

   (2) *Change letter-category of letter 'a' to 'x'*
   Change letter-category of letter 'b' to 'x'
   Change letter-category of letter 'c' to 'x'

   Likewise, both of the rules shown below could describe ***abc*** $\Rightarrow$ ***cba***, but the first rule is more abstract than the second one, since the concept of *String-Position* is of greater conceptual depth than the concept of *Letter-Category*:

(1) *Swap positions of leftmost letter and rightmost letter*

(2) *Swap positions of letter 'a' and letter 'c'*

2. *The average conceptual depth of the intrinsic-clause change-descriptors.* For example, both of the rules shown below could describe ***abc*** ⇒ ***bad***, but the first one is more abstract than the second one:

   (1) *Change letter-category of leftmost letter to successor*
   Change letter-category of middle letter to predecessor
   Change letter-category of rightmost letter to successor

   (2) *Change letter-category of leftmost letter to 'b'*
   Change letter-category of middle letter to 'a'
   Change letter-category of rightmost letter to 'd'

3. *The average conceptual depth of the extrinsic-clause object-attributes.* For example, ***abc*** ⇒ ***cba*** could be described by both of the rules shown below (the first rule is based on bridges ***a***–***a*** and ***c***–***c***, while the second is based on bridges ***a***–***c*** and ***c***–***a***):

   (1) *Swap positions of leftmost letter and rightmost letter*

   (2) *Swap letter-categories of leftmost letter and rightmost letter*

   Rule (1) is more abstract than Rule (2), since the *String-Position* concept has a greater conceptual depth than the *Letter-Category* concept.[3]

[3]On the other hand, however, it could be argued that Rule (2) is actually the more abstract rule, since Rule (1) is based on mapping ***a*** to ***a*** and ***c*** to ***c***, which is a very concrete and obvious thing to do, whereas seeing ***abc*** ⇒ ***cba*** as ***a*** turning into ***c*** and vice versa amounts to a subtler (and hence more abstract) interpretation. This example shows that, in general, the mappings underlying rules also need to be taken into account when judging rule quality, as was mentioned earlier on page 97.

Succinctness

There should be pressure to create shorter, more concise rules whenever possible. Accordingly, the fewer clauses a rule has, the better. Also, the ability of rules to describe changes in terms of the components of an object—without having to refer to each component directly—powerfully assists in the creation of pithy rules. The succinctness of a rule is thus a function of:

1. *The total number of clauses in a rule.* For example, both of the rules shown below could describe ***abc*** $\Rightarrow$ ***cba***, but the first one is more succinct than the second:

   (1) *Swap letter-categories of leftmost letter and rightmost letter*

   (2) *Change letter-category of leftmost letter to 'c'*
   Change letter-category of rightmost letter to 'a'

2. *The degree to which changes are described in terms of the components of objects.* For example, both of the rules shown below could describe ***abc*** $\Rightarrow$ ***aabbcc***, but the first one is more succinct than the second:

   (1) *Increase lengths of all objects in string by one*

   (2) *Increase length of leftmost letter by one*
   Increase length of middle letter by one
   Increase length of rightmost letter by one

   As another example, both of the rules shown below could describe ***eqe*** $\Rightarrow$ ***qeq***, but the first is more succinct than the second, even though both rules consist of just one extrinsic-clause:

(1) *Swap letter-categories of all objects in string*

(2) *Swap letter-categories of leftmost letter, middle letter, and rightmost letter*

Finally, for completeness, two other special cases should be mentioned. The identity rule *Don't change anything* is maximally uniform, maximally abstract, and maximally succinct. Verbatim rules, such as *Change string to "abd"*, are maximally uniform, minimally abstract, and maximally succinct.

3.3.6 The rule-abstraction process in detail

Metacat's structure-building processes—in keeping with the idea of the parallel terraced scan—are broken up into sequences of smaller steps carried out by chains of codelets. Thus the creation of a rule, like that of any other type of Workspace structure, is accomplished in several stages. First, a *Rule-scout* codelet examines the bridges built between the initial string and the modified string. Based on the slippages underlying these bridges, the codelet tries to "abstract out" a high-level description of how the initial string changes. If such a description can be found, it is proposed as a new rule, although at this stage the accuracy of the rule cannot be guaranteed. There is also some (small) chance that the codelet will simply ignore the bridges and instead propose a verbatim rule, bypassing the abstraction process entirely.

In any case, the newly proposed rule is subsequently evaluated by a *Rule-evaluator* codelet, which determines (1) whether or not the rule actually works (that is, whether applying the rule to the initial string really does produce the modified string), and (2) whether the rule is of high enough quality to actually merit building. If the proposed rule passes these tests, it then gets built by a *Rule-builder* codelet.[4]

[4]Unlike in Copycat, building a new rule in Metacat does not require that a previously existing rule be destroyed. Instead, many different rules can coexist in Metacat's Workspace, although only

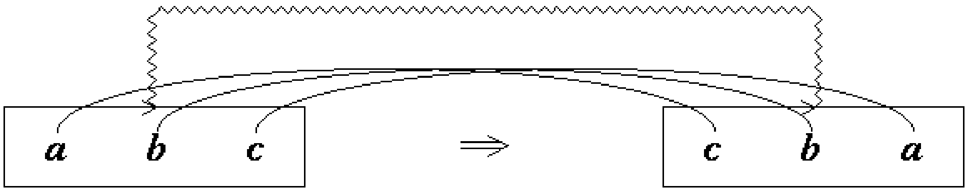


Figure 3.5: *A possible mapping for* ***abc*** ⇒ ***cba***

The first stage in this process—proposing a rule based on the slippages underlying the horizontal bridges—is fairly intricate, because in general any given set of slippages can be described in many different ways. Taken individually, every slippage represents some type of intrinsic change to an object, but it may instead be possible to describe a change extrinsically, in conjunction with other slippages. For example, the ***abc*** ⇒ ***cba*** mapping shown in Figure 3.5 involves the slippages a ⇒ *c* (supporting the ***a***–***c*** bridge) and *c* ⇒ a (supporting the ***c***–***a*** bridge). Each slippage by itself represents an intrinsic letter-category change (either to the letter-category *c* or to *a*) of a single letter, but taken together they represent an extrinsic letter-category *swap* involving two letters. These slippages could therefore give rise either to the rule

Swap letter-categories of leftmost letter and rightmost letter

consisting of a single extrinsic clause, or to the rule

Change letter-category of leftmost letter to 'c'
Change letter-category of rightmost letter to 'a'

consisting of two intrinsic clauses. In fact, several other variants of these rules are possible, depending on whether the letters ***a*** and ***c*** in ***abc*** are described in terms of their string position (*e.g.*, the *rightmost* letter) or their letter-category (*e.g.*, the letter *'c'*).

one of them will be associated with any given answer at a time.

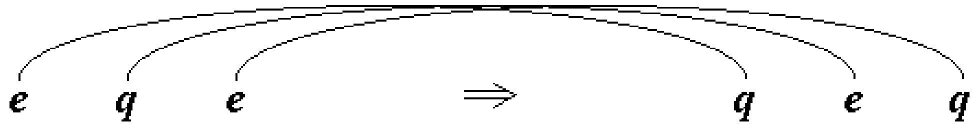


Figure 3.6: *A possible mapping for* ***eqe*** ⇒ ***qeq***

As another example illustrating a wide range of possibilities, consider the mapping ***eqe*** ⇒ ***qeq***, in which bridges ***e***–***q***, ***q***–***e***, and ***e***–***q*** are supported by the letter-category slippages $e \Rightarrow q$, $q \Rightarrow e$, and $e \Rightarrow q$ (see Figure 3.6). These slippages could be described as three separate intrinsic letter-category changes to the leftmost, middle, and rightmost letters of ***eqe***; as a combination of intrinsic and extrinsic changes (such as swapping the letter-categories of the leftmost and middle letters, and changing the letter-category of the rightmost letter to q); or as a single letter-category swap involving all three letters. In the latter case, the swap could be described either as explicitly involving the leftmost, middle, and rightmost letters of ***eqe***, or as involving all of the components of ***eqe***. Furthermore, as in the previous example, the letters ***e***, ***q***, and ***e*** could themselves be described in numerous ways (*e.g.*, ***q*** could be described as the letter *'q'* instead of the *middle* letter). All of these possible ways of constructing a rule from the bridges shown in Figure 3.6 must be *potentially* discoverable by the program; none should be excluded in principle. Thus, the rule abstraction process is necessarily stochastic.

Another factor that complicates the creation of rules is the fact that some slippages may be redundant or "misleading", and should therefore be disregarded when abstracting a rule. For example, Figure 3.7 shows a mapping for ***abc*** ⇒ ***abcd*** in which the leftmost and rightmost letters of ***abc*** map, respectively, to the leftmost and rightmost letters of ***abcd***, and the length-three successor group ***abc*** maps as a whole to the length-four successor group ***abcd***. The latter bridge is supported by the slippage *three* ⇒ *four*, which serves as the basis for constructing the rule *Increase*

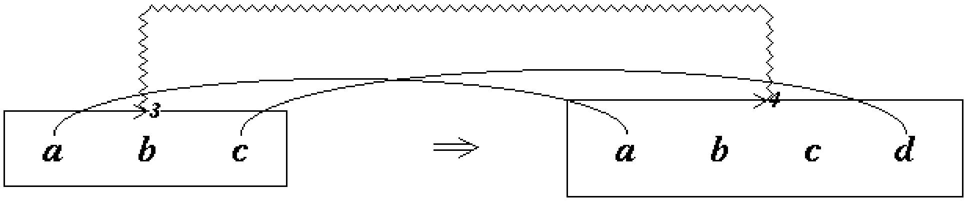


Figure 3.7: *A possible mapping for* ***abc*** $\Rightarrow$ ***abcd***

length of string by one. However, the ***c***–***d*** bridge is supported by the slippage $c \Rightarrow d$, which is, in a sense, already "implied" by the *three* $\Rightarrow$ *four* length slippage. Including the extra clause *Change letter-category of rightmost letter to successor*, based on an intrinsic change arising from the slippage $c \Rightarrow d$, would result in an incorrect rule, so this slippage should be ignored.

In general, *Rule-scout* codelets rely on a set of heuristics to help them avoid incorporating redundant or spurious changes into a rule. These heuristics increase the likelihood that a proposed rule correctly describes the differences between the initial string and the modified string—although this is not guaranteed to be the case, because the heuristics eventually break down if the horizontal mapping becomes too complicated. However, even if an errant rule manages to slip by the heuristics in the initial stage of rule creation, it will still be detected—and eliminated—by a *Rule-evaluator* codelet at the next stage.

It is important to stress that the heuristics used by *Rule-scout* codelets are not essential to the rule-creation process in principle. Rather than relying on heuristics to eliminate redundant changes, a possible alternative approach might be to simply eliminate some of the changes on a probabilistic basis. This approach would rely completely on the subsequent evaluation stage to filter out spurious rules. Eventually, if a rule were proposed with just the right combination of changes that enabled it to correctly describe the transformation of the initial string into the modified string, it would survive the evaluation stage and actually get built. However, this more "bottom-up" approach would significantly slow down the rule-creation process as mappings between strings became more complex. More and more time would be spent proposing (and then rejecting) rules that were doomed to failure from the start. The space of potentially discoverable rules would nevertheless remain the same. Thus the use of heuristics speeds up the rule-discovery process but does not represent a fundamental increase in computational power beyond that provided by the parallel terraced scan.

A step-by-step outline of the abstraction process

To abstract a rule from a set of bridges, a *Rule-scout* codelet first creates a heterogeneous mixture of intrinsic and extrinsic changes based on the bridges' supporting slippages. At the outset, all possible intrinsic changes specified by individual slippages are included in the mix. Next, if extrinsic changes are possible, they get thrown into the mix on a probabilistic basis. In addition, if "component" versions of intrinsic or extrinsic changes are possible (*i.e.*, changes that refer to all of the *components* of an object), they also get thrown in probabilistically. Redundant changes are then filtered out of the resulting mixture through the application of rule-abstraction heuristics acting as a kind of "sieve", leaving behind a final set of intrinsic and extrinsic changes. These changes are then used to create the actual set of rule clauses making up the new rule. A more detailed outline of this process is given below, along with examples illustrating each step.

1. An initial set of intrinsic changes is created from the individual slippages underlying all of the horizontal bridges. For example, a total of seven intrinsic changes would be created from the ***abc*** ⇒ ***ccbbaa*** mapping shown in Figure 3.8: one describing the string ***abc*** as reversing direction (based on the *right* ⇒ *left* slippage underlying the top-level bridge); three describing the letters ***a***, ***b***, and ***c***

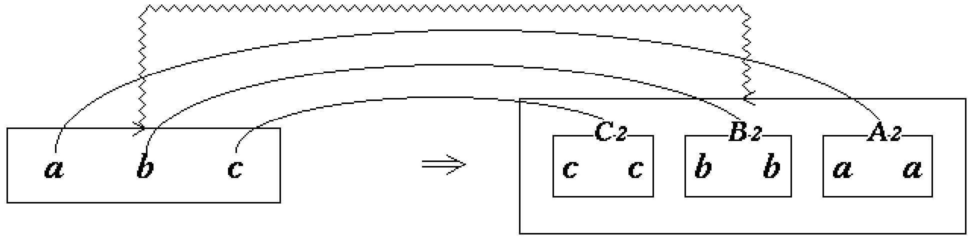


Figure 3.8: *A possible mapping for* ***abc*** ⇒ ***ccbbaa***

as each changing from a letter to a group (based on the *letter* ⇒ *group* slippages underlying the bridges ***a***–***aa***, ***b***–***bb***, and ***c***–***cc***); and three describing the letters as each changing in length from one to two (based on *one* ⇒ *two* slippages underlying the bridges ***a***–***aa***, ***b***–***bb***, and ***c***–***cc***). The bridges ***a***–***aa*** and ***c***–***cc*** are also supported by the string-position slippages *leftmost* ⇒ *rightmost* and *rightmost* ⇒ *leftmost*, but since changes to an object's position cannot be described intrinsically, these slippages do not contribute anything at this stage.

2. A few randomly-chosen subsets of bridges are examined in order to see whether any symmetries exist among the underlying slippages. If a symmetry is detected, an extrinsic change describing this symmetry is added to the current set of changes with some probability. For example, if the bridges ***a***–***aa*** and ***c***–***cc*** in Figure 3.8 happen to be examined together, the symmetric slippages *leftmost* ⇒ *rightmost* and *rightmost* ⇒ *leftmost* will be noticed. Based on these slippages, an extrinsic change describing a string-position swap between the letters ***a*** and ***c*** in ***abc*** may get created. On the other hand, if a different set of bridges were examined (such as ***a***–***aa*** and ***b***–***bb***), no such swap would be detected.

3. An extrinsic change that happens to involve all of the components of some higher-level object can be described either concretely in terms of the individual

components themselves, or more abstractly in terms of the higher-level object. Whenever such an extrinsic change is created, the level of abstraction used to describe the relevant objects is chosen probabilistically. For example, in Figure 3.6, an extrinsic letter-category swap involving the letters ***e***, ***q***, and ***e*** of ***eqe*** might get created on the basis of symmetric $e \Rightarrow q$ and $q \Rightarrow e$ slippages shared between all three bridges. The three letters of ***eqe*** could either be described individually (perhaps resulting in the rule *Swap letter-categories of leftmost letter, middle letter, and rightmost letter*), or collectively, as the components of the string ***eqe*** (resulting in the rule *Swap letter-categories of all objects in string*).

4. It is also possible for *intrinsic* changes to collectively refer to the components of objects. Sets of bridges that are "anchored" to all of an object's components are examined to see whether they have any slippage patterns in common. If a common pattern is found, an intrinsic change describing this pattern is added to the current set of changes with some probability. For example, in Figure 3.8, the bridges ***a***–***aa***, ***b***–***bb***, and ***c***–***cc***, which span the components of the string ***abc***, all share the slippages *letter* ⇒ *group* and *one* ⇒ *two*. Accordingly, two intrinsic changes may get created: one describing all of the components of ***abc*** as changing from letters to groups, and another describing all of ***abc***'s components as changing in length from one to two.

5. Once the mixture of intrinsic and extrinsic changes is complete, rule-abstraction heuristics (described more fully below) are applied to filter out the redundant changes. One example of this type of redundancy has already been discussed for the ***abc*** ⇒ ***abcd*** mapping of Figure 3.7. As another example, suppose that for the ***abc*** ⇒ ***ccbbaa*** mapping shown in Figure 3.8, an intrinsic change had been created specifying that all of ***abc***'s components change to groups (as in step 4 above). This would effectively render "obsolete" the original three intrinsic

changes (created in step 1 above) that individually describe the letters ***a***, ***b***, and ***c*** as changing to groups. Consequently, these changes would be suppressed by the heuristics.

6. The last step in the rule-abstraction process creates the actual rule clauses that appear in the proposed rule, based on the final remaining set of intrinsic and extrinsic changes. Descriptions of changed objects, and the level of abstraction used in describing the changes themselves, are chosen probabilistically as a function of the particular concepts involved. For example, suppose that for the ***abc*** ⇒ ***ccbbaa*** mapping of Figure 3.8, an extrinsic change is created that describes ***a*** and ***c*** in ***abc*** as swapping positions. The proposed rule might refer to these letters in terms of their string positions (*Swap positions of leftmost letter and rightmost letter*), or their letter-categories (*Swap positions of letter 'a' and letter 'c'*), or even both (*Swap positions of leftmost letter and letter 'c'*). Similarly, if an intrinsic change is created that describes all of ***abc***'s components as changing in length from one to two, the proposed rule might describe this either abstractly (*Increase lengths of all objects in string by one*) or literally (*Change all objects in string to groups of length two*). In both cases, the choices are biased by the conceptual depths of the relevant concepts (*i.e.*, *String-Position* versus *Letter-Category* in the first case; *successor* versus *two* in the second).

Rule-abstraction heuristics

The remainder of this section summarizes the rule-abstraction heuristics used by Metacat to create new rules. Since horizontal mappings can become arbitrarily complicated, depending on the particular strings involved, the use of these heuristics does not always result in legitimate rules being proposed, although it does greatly improve the odds in their favor.

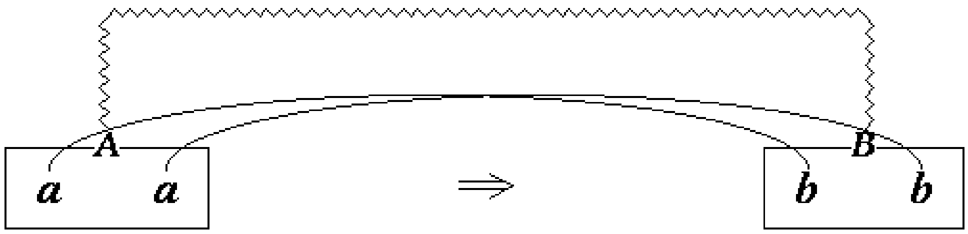


Figure 3.9: *A possible mapping for* ***aa*** ⇒ ***bb***

- If the components of an object are collectively described as all changing in some way, then these changes do not need to be described on an individual basis, and are therefore suppressed. An example of this was given earlier for the ***abc*** ⇒ ***ccbbaa*** mapping shown in Figure 3.8. Describing the components of ***abc*** as all changing to groups and all changing in length from one to two suppresses the individual *letter* ⇒ *group* and *one* ⇒ *two* changes associated with each of the letters ***a***, ***b***, and ***c***.

- If a group is described as changing its letter-category, then changes to the letter-categories of its constituent objects do not need to be individually described, and are therefore suppressed. For example, all three bridges shown in Figure 3.9 for the mapping ***aa*** ⇒ ***bb*** are supported by an $a \Rightarrow b$ letter-category slippage, and thus give rise to three separate letter-category changes: one to the ***aa*** group itself, and one to each of its letters. However, the latter two changes are redundant, since changing the letter-category of the group automatically implies changing the letter-category of each of its letters.

- If an object is described as changing its alphabetic-position (*i.e.*, from first to last, or vice versa), there is no need to describe its letter-category as changing, since this is implied by the alphabetic-position change. For example, if the ***aa*** group in the ***aa*** ⇒ ***zz*** mapping shown in Figure 3.10 is described as changing its

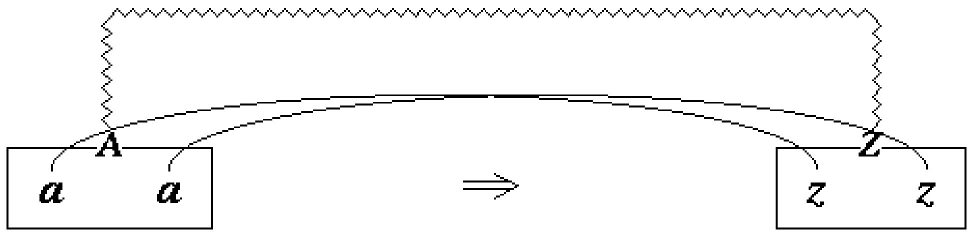


Figure 3.10: *A possible mapping for* ***aa*** ⇒ ***zz***

alphabetic-position from first to last, the group's letter-category change (based on the $a \Rightarrow z$ letter-category slippage underlying the top-level bridge) will be suppressed, as will the individual letter-category and alphabetic-position changes associated with the group's constituent letters.

- If a group is described as changing in length, then individual changes to the letter-categories of its constituent objects may in fact be spurious "side-effects" of the length change, and are therefore suppressed. An example of this was given earlier for the ***abc*** ⇒ ***abcd*** mapping shown in Figure 3.7. This mapping gives rise to two intrinsic changes: a *three* ⇒ *four* length change to the group ***abc***, and a $c \Rightarrow d$ letter-category change to the letter ***c***. The latter change, however, is suppressed, since it is really a consequence of the former change.

- If a group is described as changing to a letter, there is no need to describe the group's length as changing to one, since this is automatically implied by the group-to-letter change. The length change is therefore suppressed. For example, an ***aa*** ⇒ ***a*** bridge would give rise to two intrinsic changes to the ***aa*** sameness group: an object-type change based on the slippage *group* ⇒ *letter*, and a length change based on the slippage *two* ⇒ *one*. Only the first change, however, would be included in the final set of changes used to create the new rule.

- If several intrinsic changes happen to describe a set of objects in a way that is equivalent to some *extrinsic* change, then the intrinsic changes are suppressed. For example, in the ***eqe*** $\Rightarrow$ ***qeq*** mapping shown in Figure 3.6, an intrinsic letter-category change is created for each of the letters ***e***, ***q***, and ***e*** of ***eqe***. However, an extrinsic change describing a three-way letter-category swap among these letters may also get created—in which case the intrinsic changes are redundant, and are therefore suppressed.

3.4 Nondeterministic Rule Translation

Once a new rule has been built, it is available for *Answer-finder* codelets to use in trying to create new answers. These codelets probabilistically decide whether or not to attempt answer creation on the basis of the overall strengths of the horizontal and vertical mappings between strings[5] (as long as these mappings are weak, codelets are unlikely to try).

At any given time in Metacat's Workspace, there may be several different rules available for codelets to choose from (unlike in Copycat, where only one rule at a time can exist). Codelets choose rules probabilistically as a function of rule quality and the *degree of support* each rule currently has in the Workspace. A rule's degree of support depends on the strengths (and continued existence) of the original bridges used in creating the rule. In general, since the horizontal mapping between the initial and modified strings may change over time, bridges that played a critical role in the creation of a particular rule may no longer exist at the time the rule is chosen by an *Answer-builder* codelet—or if they do still exist, they may be very weak. Additionally, new bridges or groups may have been built that cause the chosen rule to no longer work correctly when applied to the initial string. In any case, if the situation has

[5]The strength of a mapping between two strings is a function of the average mapping-specific happiness values of the objects in the strings. These values were discussed earlier in section 3.3.1.

changed to the extent that the chosen rule either is no longer supported by strong bridges or no longer works, the codelet abandons the rule and fizzles. Otherwise, the codelet *translates* the rule from the "top" situation represented by the initial string into the analogous "bottom" situation represented by the target string, using the slippages underlying the vertical mapping between the initial string and target string as a guide. The translated rule is then applied to the target string, as in Copycat, yielding either a new answer or a "snag" condition.

The rule-translation process in Metacat is more complicated than in Copycat, due to the greater structural complexity of Metacat's rules, but the basic principles are the same. The slippages supporting the vertical bridges are applied to the concepts making up the rule, causing some of them to "slip" to new concepts. However, in Metacat this process is *nondeterministic*, whereas in Copycat it is deterministic. (Copycat's decision whether or not to translate a rule is made probabilistically, but the translation process itself is deterministic.) This allows a greater degree of "sloppiness" on the part of the program when coming up with answers, as illustrated in the following examples.

One way of interpreting the problem *"**abc** ⇒ **aabbcc**; **iijjkk** ⇒ **?**"* is to regard all of the letters of ***abc*** as increasing their lengths by one. If the sameness groups ***ii***, ***jj***, and ***kk*** are seen as the "letters" of ***iijjkk***, the answer ***iiijjjkkk*** suggests itself. Likewise, the problem *"**abc** ⇒ **aabbcc**; **kkjjii** ⇒ **?**"* can be viewed in the same way, suggesting the answer ***kkkjjjiii***. However, here the pressures are somewhat different, since ***abc*** is a successor group but ***kkjjii*** is a *predecessor group* (assuming that the strings are both seen as going to the right). Perhaps the ***kk***, ***jj***, and ***ii*** groups should instead *decrease* their lengths by one, in accordance with the successor–predecessor symmetry, yielding the answer ***kji***. Figures 3.11 and 3.12 show configurations of Metacat's Workspace that represent these two situations. Both configurations involve exactly the same string mappings, but exhibit different answers due to the

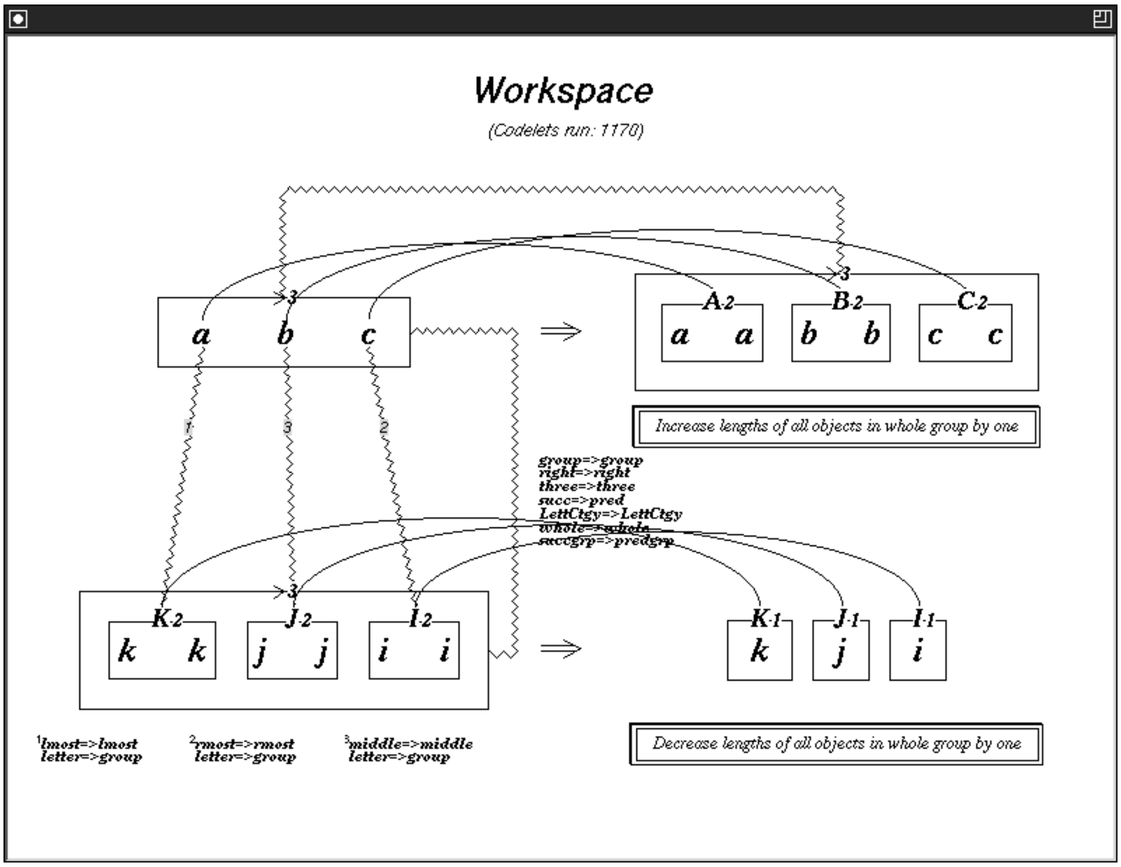


Figure 3.11: *Applying the* successor ⇒ predecessor *slippage when translating the rule yields the answer* ***kji***.

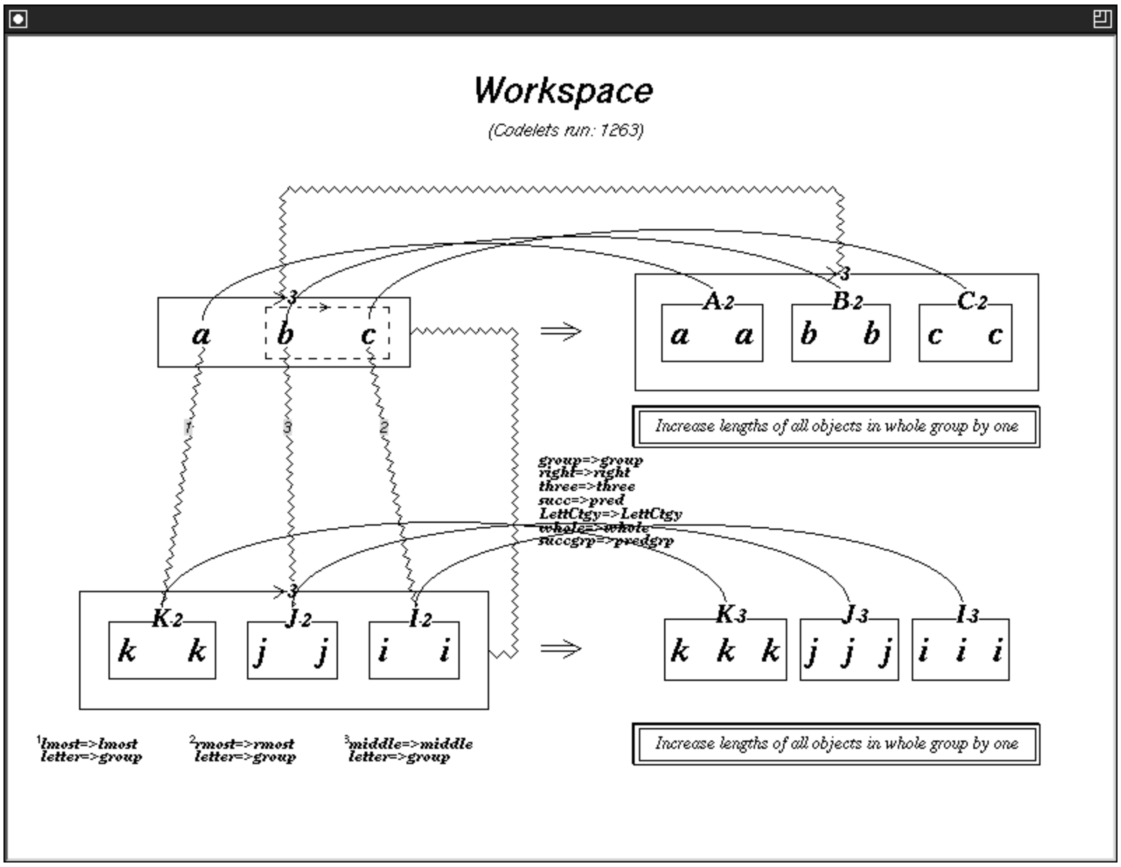


Figure 3.12: *Ignoring the successor ⇒ predecessor slippage when translating the rule yields the answer* ***kkkjjjiii***.

probabilistic nature of rule translation. In the first case, the *successor ⇒ predecessor* slippage supporting the top-level vertical bridge was applied to the rule, resulting in the translated rule *Decrease lengths of all objects in string by one* (*i.e.*, change the lengths of objects to their predecessor instead of successor), while in the second case this slippage was not applied.

As another example, Figures 3.13 and 3.14 show two essentially identical Workspace configurations for the problem *"**abc** ⇒ **cba**; **mrrjjj** ⇒ ?"*, in which ***mrrjjj*** is seen as a *1–2–3* successor group corresponding to ***abc***, and the letters ***a*** and ***c*** of ***abc***

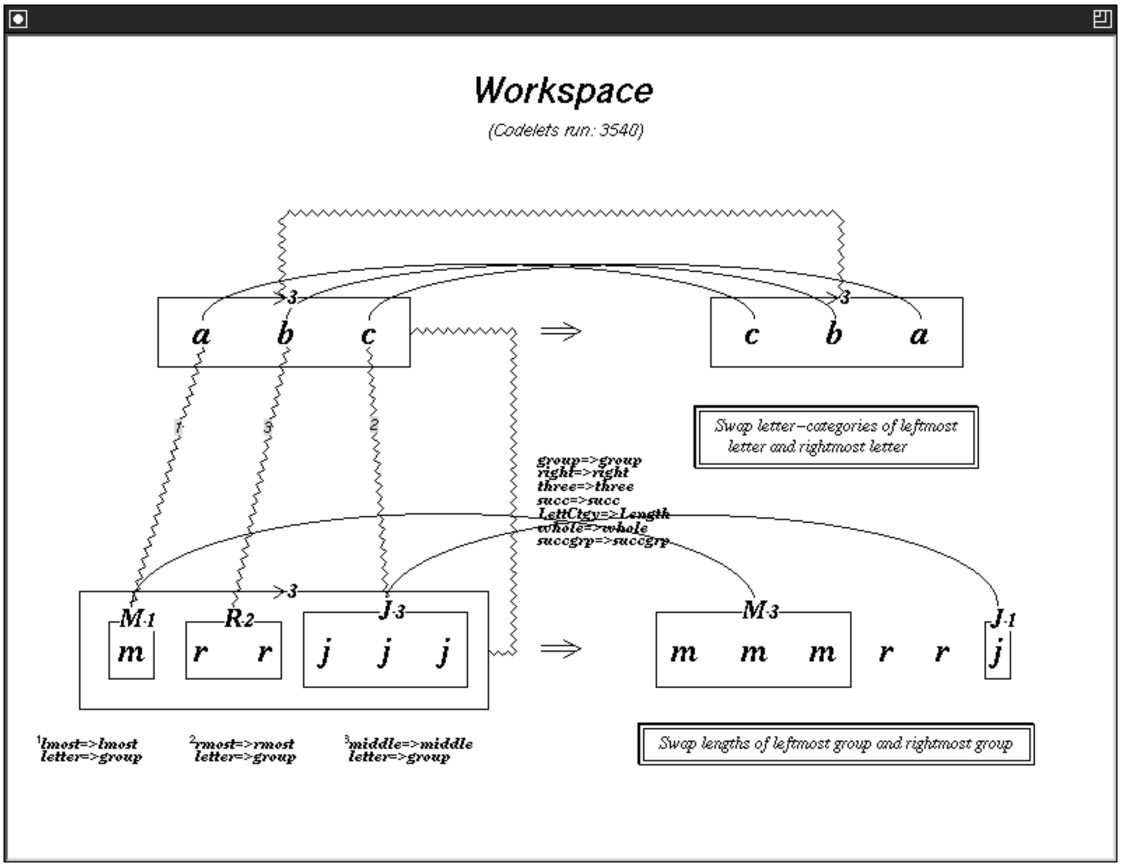


Figure 3.13: *Applying the* Letter-Category ⇒ Length *slippage when translating the rule yields the answer* ***mmmrrj***.

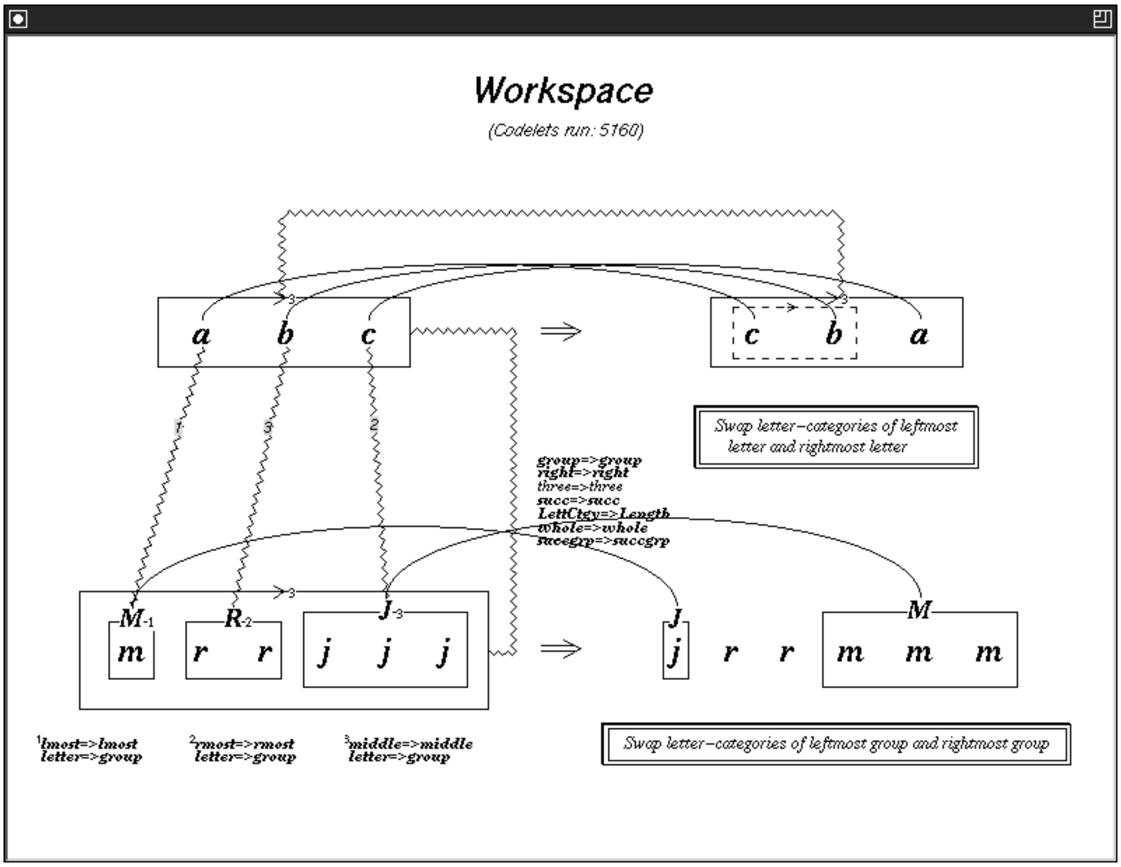


Figure 3.14: *Ignoring the Letter-Category ⇒ Length slippage when translating the rule yields the answer* ***jrrmmm****.*

are interpreted as swapping their letter-categories (as opposed to their string positions). In the first case, the *Letter-Category* ⇒ *Length* slippage is taken into account when translating the rule, resulting in the *lengths* of group ***m*** and group ***jjj*** being swapped. In the second case, this slippage is ignored, resulting in the *letter-categories* of ***m*** and ***jjj*** being swapped.

3.4.1 Coattail slippages

There is another way in which Metacat's rule translation process is nondeterministic, apart from probabilistically ignoring certain vertical bridge slippages. This has to do with slippages that get pulled along on the "coattails" of other, related slippages. In Copycat, no such coattail-slippage mechanism exists, although Mitchell notes that it would be a desirable extension of the program [Mitchell, 1993, page 192]. A simple example that illustrates this idea is the problem *"**a** ⇒ **b**; **z** ⇒ **?**"*. A natural answer is ***y***, based on the notion that since ***a*** changes to its successor and ***z*** is, in an alphabetic sense, the opposite of ***a***, one should do the "opposite thing" to ***z*** and change it to its *predecessor*. This answer makes all the more sense given the fact that taking the successor of ***z*** is impossible. Unfortunately, Copycat cannot get this answer, because using the slippage *successor* ⇒ *predecessor* in translating the rule can only be done if that slippage supports some vertical bridge. In this problem, the only potential slippage between ***a*** and ***z*** is *first* ⇒ *last*; the concepts *successor* and *predecessor* never come up, so there is no way for the translated rule to specify changing ***z*** to its predecessor instead of its successor.

In Metacat, however, applying a *first* ⇒ *last* slippage to the *successor* concept in some rule may occasionally cause the concept to slip to *predecessor*, even though the slippage explicitly specifies slipping only *first* to *last*. The reason for this is that the *successor* concept is linked to *predecessor* in the Slipnet via a link labled by the concept *opposite*, which also characterizes the *first* ⇒ *last* slippage. If *opposite* is

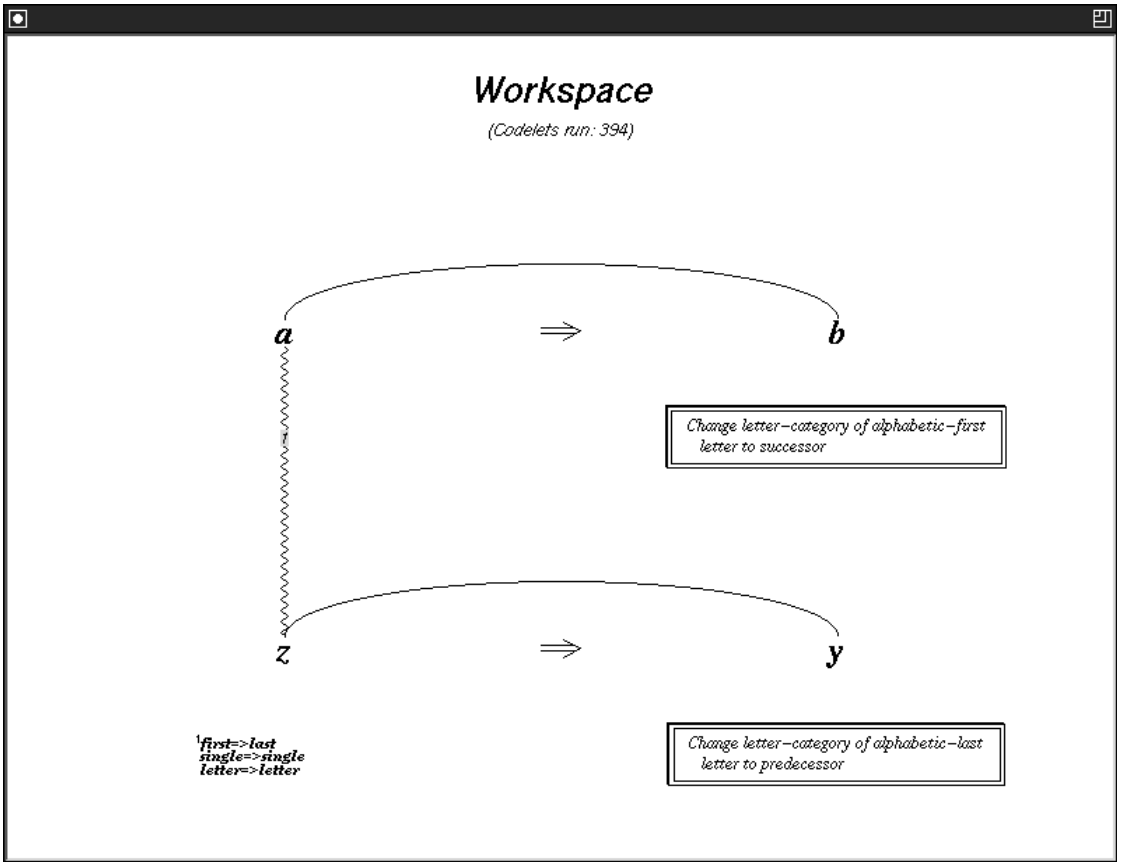


Figure 3.15: *An example of rule translation in which a* first ⇒ last *slippage causes both the concept* first *and the concept* successor *to slip to their opposite concepts.*

strongly activated, the distance between *successor* and *predecessor* shrinks, making a *successor* ⇒ *predecessor* slippage more likely in the presence of any type of *opposite* slippage. Thus, it is indeed possible for Metacat to get the answer ***y*** to the above problem (see Figure 3.15). In general, a slippage labeled by a highly activated concept (such as *opposite*) may cause slippages of a similar type to occur between other concepts that are not explicitly present in the original slippage (but that are related in the same way). The probability of such coattail slippages occurring is a function of the activation of the original slippage's label node.

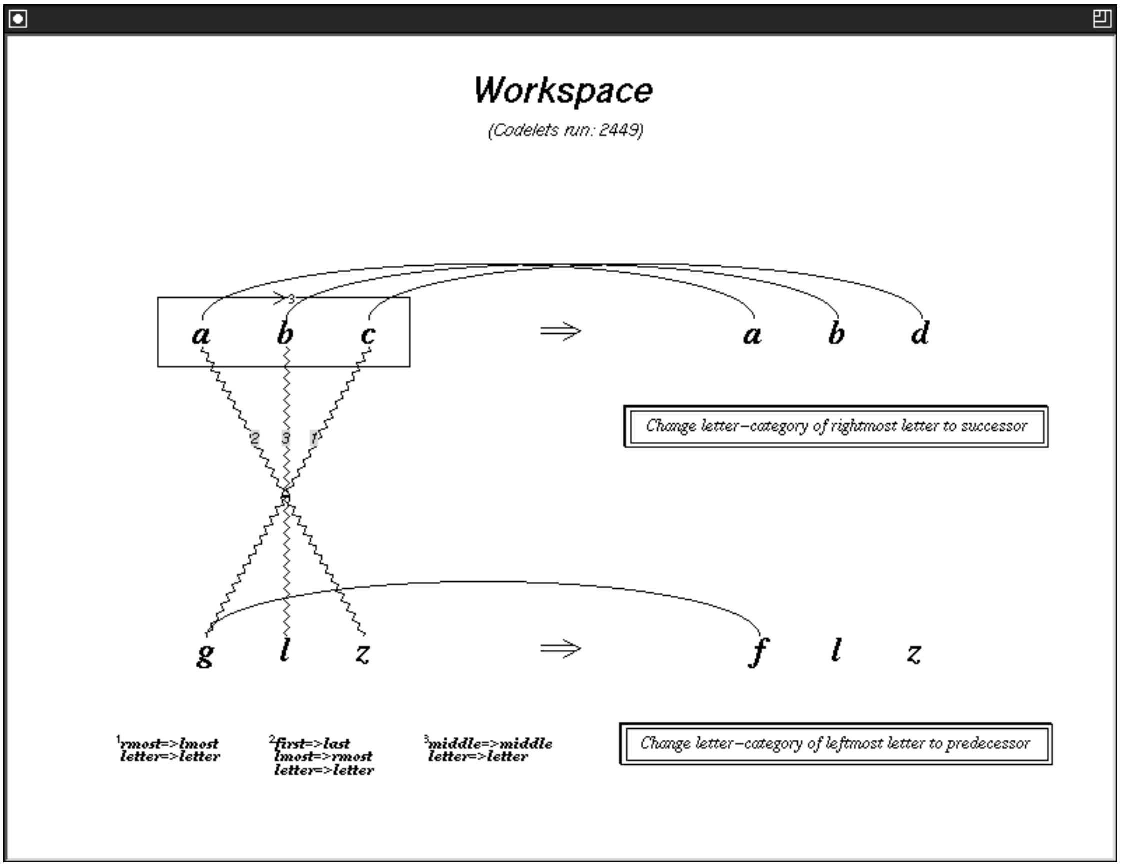


Figure 3.16: *Another example illustrating the "coattail slippage" effect.*

As another example, consider the problem *"**abc** ⇒ **abd**; **glz** ⇒ **?**"*, discussed by Mitchell. Even if Copycat notices the alphabetic ***a*–*z*** symmetry in this problem and consequently maps the rightmost letter ***c*** of ***abc*** to the leftmost letter ***g*** of ***glz***, it cannot answer ***flz***, since there is no possibility of seeing ***glz*** as a left-going predecessor group, which would give rise to a *successor ⇒ predecessor* slippage. The best it can do is to answer ***hlz***. Metacat, on the other hand, can sometimes make the *successor ⇒ predecessor* slippage on the coattails of the *first ⇒ last* slippage, yielding the answer ***flz*** (see Figure 3.16).

3.5 Other Refinements to Copycat

Before closing this chapter, it is worth mentioning one other refinement that has been made to the bridge-building mechanisms inherited from Copycat. In Metacat, as in Copycat, the strength of a bridge depends on both the *internal coherence* of the bridge and the *mutual support* it receives from other existing bridges[6] (see [Thagard, 1989] for a discussion of similar ideas that arise in the context of scientific theorizing). The internal coherence and mutual support of bridges reflect the degree to which the underlying concept-mappings support each other. Two concept-mappings support each other if they involve the same relationship (such as *opposite*) and if their corresponding descriptors are linked in the Slipnet.

For example, the concept-mappings *first* ⇒ *last* and *leftmost* ⇒ *rightmost* support each other, since both concept-mappings are labeled by the *opposite* relation, and the corresponding descriptor-pairs *first* and *leftmost* (as well as *last* and *rightmost*) are linked in the Slipnet. A bridge based on these two concept-mappings would therefore be internally coherent, such as an ***a***–***z*** bridge in the problem shown below:

abc	⇒	***abd***
xyz	⇒	***?***

Likewise, a symmetric ***c***–***x*** bridge based on *rightmost* ⇒ *leftmost* would reinforce the ***a***–***z*** bridge, since *rightmost* ⇒ *leftmost* supports (and is supported by) the concept-mappings *leftmost* ⇒ *rightmost* and *first* ⇒ *last*.

On the other hand, Copycat considers the concept-mapping *first* ⇒ *first* to be *incompatible* with *leftmost* ⇒ *rightmost*, which causes problems in certain situations. For example, consider the problem shown below:

[6] This is true for both horizontal and vertical bridges in Metacat, but only for vertical bridges in Copycat. See [Mitchell, 1993, Chapter 3] for a more complete description of bridge strength.

abc $\Rightarrow$ ***abd***
cba $\Rightarrow$ ***?***

If the ***a*** letters get described as alphabetic-first letters, then a diagonal ***a***–***a*** bridge based on the concept-mappings *first* $\Rightarrow$ *first* and *leftmost* $\Rightarrow$ *rightmost* is considered to be internally *incoherent*, and is therefore very difficult to build. Furthermore, such a bridge cannot coexist with a diagonal ***c***–***c*** bridge, since the latter bridge would be based on *rightmost* $\Rightarrow$ *leftmost*, which conflicts with the *first* $\Rightarrow$ *first* concept-mapping underlying ***a***–***a***.

In order to remedy this problem in Metacat, the notion of incompatibility between concept-mappings has been refined. Metacat's Slipnet includes labels on four links that were previously unlabeled in Copycat (see Figure 3.17). These links, labeled by either *identity* or *opposite*, allow Metacat to make finer distinctions between concept-mappings. (This approach, of course, is not completely satisfactory, since the concepts *leftmost* and *rightmost* are clearly neither identical to—nor exactly the opposite of—the concepts *left* and *right*, but it nevertheless serves as a reasonable interim solution to the problem.) In order to be incompatible, two concept-mappings must not only be based on different relationships, but the way in which their corresponding descriptor-pairs are linked must also differ. For example, the concept-mappings *leftmost* $\Rightarrow$ *rightmost* and *right* $\Rightarrow$ *right* are incompatible, since (1) they are based on different relationships (*i.e.*, *opposite* versus *identity*) and (2) the concepts *leftmost* and *right* are linked in a different way than the concepts *rightmost* and *right* (*i.e.*, the former pair is linked by *opposite*, while the latter pair is linked by *identity*). These concept-mappings are also incompatible in Copycat, but for reason (1) only.

In contrast, *leftmost* $\Rightarrow$ *rightmost* and *first* $\Rightarrow$ *first* are *not* incompatible in Metacat, even though one is based on *opposite* and the other on *identity*, since *leftmost* and *first* are linked in the same way as *rightmost* and *first* (*i.e.*, both links are unlabeled).

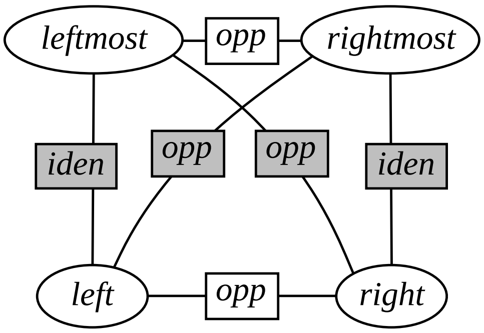


Figure 3.17: *A portion of Metacat's Slipnet showing the links between the concepts leftmost, rightmost, left, and right. New link labels are shown in grey (compare with Figure 1.1 on page 19). Aside from these four new labels, Metacat's Slipnet is exactly the same as Copycat's Slipnet.*

For the same reason, *first* ⇒ *last* does not conflict with either *leftmost* ⇒ *leftmost* or *rightmost* ⇒ *rightmost*, unlike in Copycat. Figure 3.18 shows a situation in which these latter concept-mappings simultaneously support a set of vertical bridges without conflicting.

Although the above concept-mappings are not incompatible with each other in Metacat, neither are they mutually supporting. For two concept-mappings to support each other, they must be based on the same relationship, and their corresponding descriptor-pairs must be linked in the same way. For example, *first* ⇒ *last* and *leftmost* ⇒ *rightmost* are mutually supporting (as they are in Copycat), since both are based on the *opposite* relation, and the concepts *first* and *leftmost* are linked in the same way as *last* and *rightmost* (*i.e.*, both links are unlabeled). Likewise, *leftmost* ⇒ *rightmost* and *right* ⇒ *left* support each other, since the link between *leftmost* and *right* is labeled in the same way as the link between *rightmost* and *left* in Metacat's Slipnet (*i.e.*, they are both *opposite* links).

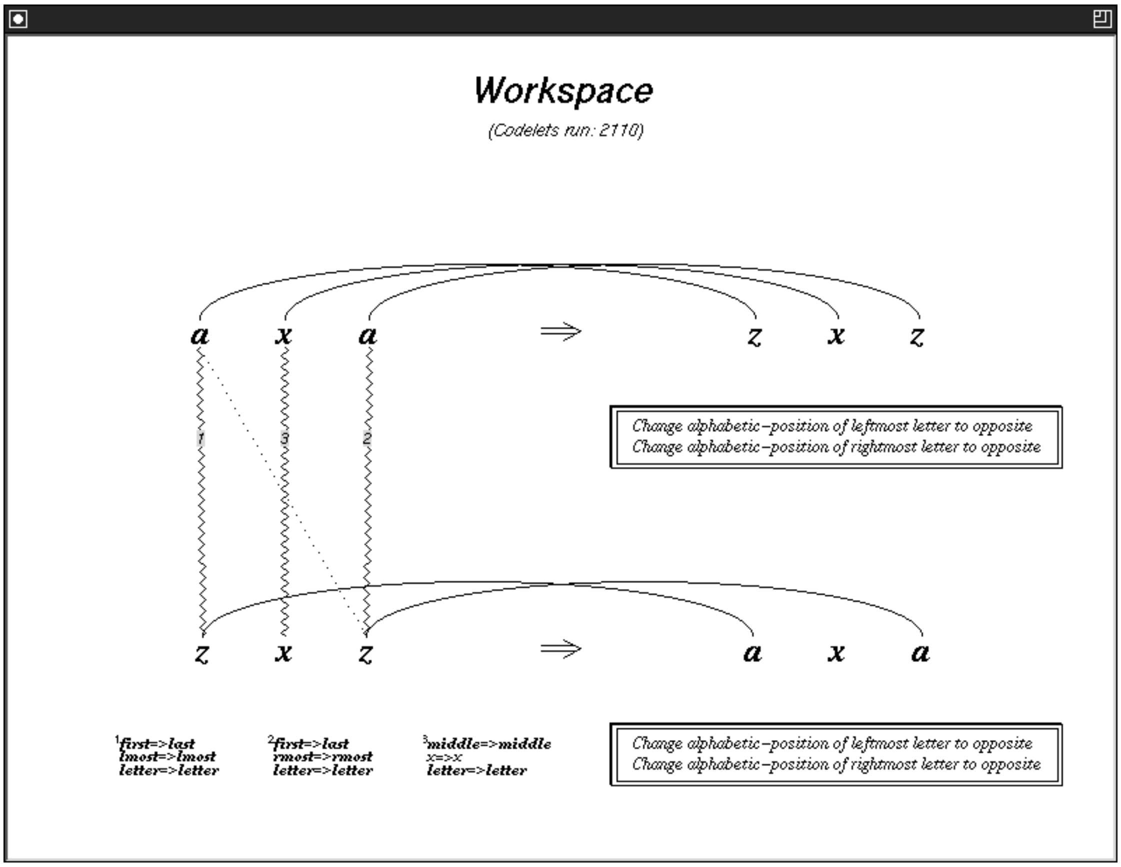


Figure 3.18: *A possible configuration of Metacat's Workspace for the problem "**axa** ⇒ **zxz**; **zxz** ⇒ **?**", in which the vertical bridges are supported by the non-conflicting concept-mappings* first ⇒ last, leftmost ⇒ leftmost, *and* rightmost ⇒ rightmost.

CHAPTER FOUR

An Architecture for Self-Watching

This chapter discusses in detail Metacat's new architectural components for self-watching, which together provide a common unifying framework in which to address the various objectives of the Metacat project outlined earlier in Chapter 2. These objectives—making the program sensitive to patterns in its own processing, giving the program the ability to remember its answers and to be reminded of answers it has previously encountered, getting the program to compare and contrast different answers, and giving the program the ability to work backwards from a given answer to a coherent justification of that answer—focus on extending the capabilities of Copycat in different ways, but all share the central, overarching goal of imbuing the program with a deeper sense of "awareness" of what it is doing (and why) as it deals with analogy problems in its microworld.

The first section describes themes and the Themespace in detail, and discusses the critical role played by themes in regulating top-down pressures in Metacat. The next section introduces the general notion of a *pattern*, another central idea of Metacat. This is followed by a discussion of how themes (and patterns of themes) enable the program to work backwards from an answer provided by the user, in an effort to discover why the answer makes sense. The next two sections discuss the Temporal

Trace—the focal point of Metacat's "self-awareness"—and the way in which the program uses the information stored there to control its own behavior. Finally, the last section describes Metacat's long-term Episodic Memory—where abstract characterizations of answers are stored—and the way in which these answers can be compared and contrasted by the program on the basis of their abstract similarities and differences. Detailed sample runs of the program illustrating all of these capabilities will be presented in the following chapter.

4.1 Themes and the Themespace

As was explained in the overview of the Metacat architecture given in section 2.4 of Chapter 2, themes are structures that represent key ideas underlying an answer to an analogy problem. More specifically, they represent regularities among the *bridges* that make up the mappings between strings. For example, in the problem "***abc*** ⇒ ***abd***; ***ijk*** ⇒ ***?***", if the strings ***abc*** and ***ijk*** are mapped onto each other by the vertical bridges ***a***–***i***, ***b***–***j***, and ***c***–***k***, these bridges will be supported by the concept-mappings *leftmost* ⇒ *leftmost*, *middle* ⇒ *middle*, and *rightmost* ⇒ *rightmost* (among others). These three concept-mappings are all based on the idea of string-position identity, and thus can be represented by a single vertical theme composed of the Slipnet concepts *String-Position* and *identity*. This particular theme would be associated with any answer that depended on seeing ***abc*** and ***ijk*** as going in the same direction—such as ***ijl*** or ***ijd***. Likewise, if an answer depended on seeing ***abc*** and ***abd*** as going in the same direction, represented by the horizontal bridges ***a***–***a***, ***b***–***b***, and ***c***–***d*** (that is, if the answer's *rule* depended on this), then a *horizontal*[1] *String-Position:identity* theme would also be associated with the answer—again, as in the case of ***ijl*** or ***ijd***.

[1]More precisely, a *top* theme, since horizontal bridges between the initial string and the modified string are being described.

As another example, the answer ***kkjjii*** to the problem *"**abc** ⇒ **ccbbaa**; **ijk** ⇒ ?"* depends on seeing ***abc*** and ***ijk*** as going in the same direction—based, as before, on the vertical bridges ***a***–***i***, ***b***–***j***, and ***c***–***k***—and on seeing ***abc*** and ***ccbbaa*** as going in opposite directions, based on the horizontal bridges ***a***–***aa*** and ***c***–***cc*** (supported by the slippages *leftmost ⇒ rightmost* and *rightmost ⇒ leftmost*). Consequently, the vertical theme *String-Position: identity* and the horizontal theme *String-Position: opposite* would be associated with this answer. In addition, the *one ⇒ two* slippages supporting the horizontal bridges ***a***–***aa***, ***b***–***bb***, and ***c***–***cc*** would give rise to the horizontal theme *Length: successor*. Other themes are also possible. In the case of *"**abc** ⇒ **abd**; **ijk** ⇒ ?"*, *letter ⇒ letter* concept-mappings underlie both the vertical and the horizontal bridges, so vertical and horizontal *Object-Type: identity* themes would also be associated with the answers ***ijl*** or ***ijd***. In the case of *"**abc** ⇒ **ccbbaa**; **ijk** ⇒ ?"*, *letter ⇒ letter* concept-mappings underlie the vertical bridges, while *letter ⇒ group* slippages underlie the horizontal bridges, so a vertical *Object-Type: identity* theme and a horizontal *Object-Type: different* theme would be associated with the answer ***kkjjii***.

In general, *vertical* themes describe vertical bridges between the initial string and the target string, *top* themes describe horizontal bridges between the initial string and the modified string, and *bottom* themes describe horizontal bridges between the target string and the answer string (which get built when Metacat runs in "justify mode", working backwards from a given answer to an interpretation of the answer). As will become clearer later, differentiating between vertical, top, and bottom themes enables Metacat to selectively focus top-down pressure on specific types of Workspace structures (for example, on bridges between the initial string and the modified string). In contrast, top-down forces in Copycat are often too diffuse, too lacking in specificity to enable the program to build the types of structures it most needs to build at a particular moment, as Mitchell has pointed out [Mitchell, 1993, Chapter 7].

For example, if the Slipnet concept *successor-group* becomes highly activated, it will exert top-down pressure to build successor groups wherever possible, with no preference given to any particular string, even if such groups are really only needed in one string. Codelets may thus waste considerable time and effort looking for successor groups in all the wrong places, making it less likely that they will be built where they are really needed. Likewise, if the *opposite* concept is active, it will promote the creation of bridges anywhere that are based on slippages labeled by the *opposite* concept. (Of course, in Copycat only bridges between the initial string and the target string can be built in general, so the distinction between different mappings does not arise.) In any case, themes in Metacat are associated with particular mappings between strings, which makes them more effective than individual Slipnet concepts in channeling top-down pressure in specific directions.

Regardless of its type, every theme consists of a *category*, which can be any Slipnet category node, such as *String-Position*, *Letter-Category*, or *Object-Type*[2], and a *relation*, which can be any Slipnet relation node (*i.e.*, any node that can be used to label links between concepts in the Slipnet), such as *opposite*, *successor*, or *identity*, or else the absence of a specific concept, which represents the idea of *different*. The particular combination of theme-type, category, and relation uniquely identifies every theme. All in all, a total of 66 distinct themes are possible.

Whenever a new bridge is built between two Workspace structures, themes based on the concept-mappings underlying the bridge get created and added to the Themespace, if they are not already present. For example, if a vertical bridge is built between the letters ***a*** and ***z*** in the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*, based on the concept-mappings *leftmost ⇒ rightmost*, *letter ⇒ letter*, and *first ⇒ last*, then the

[2]The theme categories *Object-Type* and *Group-Type* are alternative names for the concepts *Object-Category* and *Group-Category*.

vertical themes *String-Position: opposite*, *Object-Type: identity*, and *Alphabetic-Position: opposite* get created and added to the Themespace. These themes remain associated with the bridge for as long as the bridge exists. If a theme happens to already exist when a bridge based on the theme is built, then the existing theme is associated with the new bridge, and a new theme is not created. If a vertical ***b***–***y*** bridge, for instance, had already been built prior to the ***a***–***z*** bridge, the *Object-Type: identity* theme would already exist in the Themespace (on account of the *letter* ⇒ *letter* concept-mapping underlying the ***b***–***y*** bridge). A duplicate *Object-Type: identity* theme would not be added to the Themespace; rather, the existing theme would simply be associated with the new ***a***–***z*** bridge.

As was mentioned in Chapter 2, each theme in the Themespace has an activation level ranging between -100 and $+100$. (There is no effective difference between a theme having an activation level of zero and the theme not existing in the Themespace.) Themes receive periodic infusions of activation from the Workspace structures (*i.e.*, the bridges) associated with them, as a function of structure strength, with stronger structures sending more powerful jolts of activation to their associated themes. For instance, in the above example, the *Object-Type: identity* theme would receive activation periodically from both the ***a***–***z*** bridge and the ***b***–***y*** bridge, while the *String-Position: opposite* theme would receive activation from only the ***a***–***z*** bridge. In the absence of further infusions of activation from the Workspace, theme activations gradually decay over time, although the rate of decay does not depend on the particular concepts that make up a theme (unlike concept activations in the Slipnet, where the rate of decay depends on conceptual depth).

Generally speaking, the activation level of a theme is intended to represent how explicitly "aware" Metacat is of a particular idea in its current interpretation of an analogy problem. At any given time, many ideas are *implicitly* present in the Workspace structures making up the mappings between strings, but highly activated themes represent the *explicit* recognition, on the part of the program, of the importance of certain ideas. Put another way, the activation of a theme reflects the amount of "evidence" that exists in favor of regarding that particular idea as playing an important role in characterizing the situation at hand. Ideas represented by highly activated themes are likely to be of central importance, while ideas represented by weakly-activated themes are likely to be irrelevant.

4.1.1 Organization of the Themespace

Not all themes are compatible with each other. In general, two themes of the same type having the same category but different relations are incompatible, since each one reflects a different kind of relationship among a set of objects (with respect to some particular aspect of the objects). For example, the top themes *Letter-Category: identity* and *Letter-Category: successor* represent the mutually contradictory ideas of (1) seeing objects in the initial string and modified string as corresponding to one another on the basis of identical letter-categories, and (2) seeing objects in the strings as corresponding on the basis of letter-category successorship. In the problem "***abc*** ⇒ ***abcd***; ***ijk*** ⇒ ***?***", for instance, a natural way of viewing ***abc*** ⇒ ***abcd*** is to regard the letters ***a***, ***b***, and ***c*** in ***abc*** as corresponding to the letters ***a***, ***b***, and ***c*** in ***abcd*** (with ***d*** being the "odd letter out"). This idea, represented by a *Letter-Category: identity* theme, suggests interpreting ***abc*** as a successor group beginning with the letter ***a*** whose length increases by one.

However, an alternative (if somewhat unnatural) possibility is to regard ***a***, ***b***, and ***c*** as corresponding to ***b***, ***c***, and ***d*** in ***abcd*** on the basis of *successorship* (where ***a*** is now the odd letter). This idea, represented by a *Letter-Category: successor* theme, suggests interpreting ***abc*** as a *predecessor* group beginning with the letter ***c*** whose length increases by one, and whose starting letter also "increases by one" from ***c*** to ***d***. Like the conflicting interpretations of a Necker cube, these two interpretations of the ***abc*** ⇒ ***abcd*** change are mutually exclusive. Having both of the themes *Letter-Category: identity* and *Letter-Category: successor* highly active at the same time therefore makes little sense.

In order to discourage the emergence of inconsistent interpretations of string mappings, incompatible themes in the Themespace exert inhibitory effects on each other, in proportion to their levels of activation. More precisely, the Themespace is organized into mutually-inhibitory *clusters* of themes all sharing the same theme-type and category. For instance, the top themes shown below make up one cluster:

Letter-Category: identity
Letter-Category: successor
Letter-Category: predecessor
Letter-Category: different

If more than one theme within a cluster is positively activated, the themes in the cluster will compete among themselves for dominance, in a manner reminiscent of a "winner-take-all" network. (Themes in different clusters have no effect on each other.) A theme is *dominant* if its activation level exceeds that of all other themes in its cluster by a substantial margin (currently set to 90). The more infusions of activation a theme receives from Workspace structures that support it, the more likely it will be to eventually suppress its weaker intra-cluster competitors and become dominant, driving the other theme activations toward zero. The idea is that as more Workspace structures involving a particular theme are built (and the longer these structures persist), the more evidence there is that the theme is an important organizing motif underlying the interpretation emerging in the Workspace. Themes that accumulate enough evidence of their importance eventually gain the upper hand over other themes in their cluster, in a kind of "locking-in" process that occurs simultaneously for each cluster of themes in the Themespace. The result (it is hoped) is that a single, consistent way of viewing an analogy problem gradually emerges from a number of mutually inconsistent alternatives vying for supremacy. Furthermore, the most important ideas underlying the resulting interpretation are, in the end, represented explicitly by a set of highly-activated, dominant themes in the Themespace.

To illustrate these ideas, Figure 4.1 shows a snapshot of Metacat's Themespace during a run of the problem *"**abc** ⇒ **abd**; **kkjjii** ⇒ **?**"*, in which a total of eight themes have been created and added to the Themespace. The activations of top themes are shown in the window above the Workspace, while those of vertical themes are shown to the left. Each Themespace "panel" represents a particular cluster of themes, and is labeled by the cluster's theme category. For instance, the letter-category top themes *Letter-Category: identity* and *Letter-Category: successor* are shown in the leftmost panel of the Top Themes window, with the *identity* theme more strongly activated than the *successor* theme. This is because more top bridges have been built between objects of identical letter-categories (*i.e.*, the bridges ***a***–***a*** and ***b***–***b***) than between objects related by letter-category successorship (*i.e.*, the bridge ***c***–***d***), and thus the former theme has received more boosts of activation than the latter. However, neither theme is dominant in its cluster.

In contrast, the top themes *String-Position: identity* and *Object-Type: identity are* dominant (as indicated by the highlighted panels), since each is much more strongly activated than any competing theme within its cluster (in fact, no competing themes exist in either cluster). This is because all three top bridges are between objects of the same type (*i.e.*, letters), all of which share identical positions in their respective strings. Consequently, all three bridges contribute activation to the same *Object-Type* and *String-Position* themes.

In the case of the vertical mapping between ***abc*** and ***kkjjii***, the vertical themes *Letter-Category: different* and *String-Position: identity* are dominant, reflecting the fact that all of the vertical bridges map objects of different letter-categories and

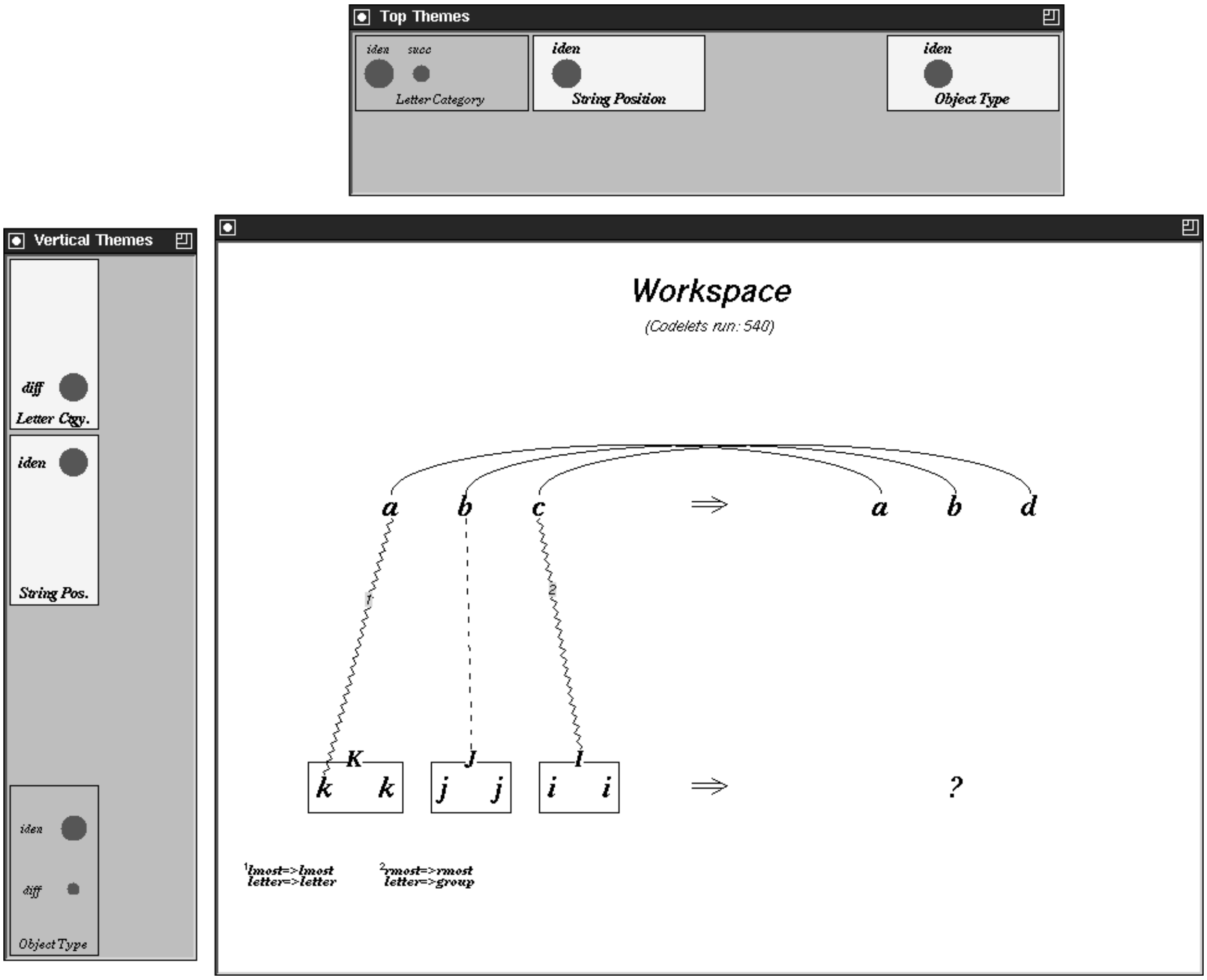


Figure 4.1: *The state of Metacat's Themespace during a run of the problem "**abc**⇒ **abd**; **kkjjii**⇒ **?**", showing the activations of various themes. Top themes, characterizing the horizontal mapping, are shown in the window above the Workspace, while vertical themes are shown to the left. Dominant themes are highlighted.*

identical string positions onto each other. The *Object-Type* cluster, however, contains competing *identity* and *different* themes, reflecting the mixture of *letter* ⇒ *letter* and *letter* ⇒ *group* concept-mappings underlying the vertical bridges. Consequently, no dominant vertical *Object-Type* theme exists.

Figure 4.2 shows the same run after Metacat has found the answer ***kkjjhh***. The final set of dominant themes consists of the two top themes *String-Position: identity* and *Object-Type: identity*, and the four vertical themes *Letter-Category: different*, *String-Position: identity*, *Group-Type: opposite*, and *Direction: identity*. The latter two themes reflect the fact that Metacat has perceived ***abc*** and ***kkjjii*** as groups of opposite types (*i.e.*, ***abc*** as a predecessor group and ***kkjjii*** as a successor group) going in the same direction (*i.e.*, both to the left). The vertical *Object-Type: different* theme remains just below the dominance threshold, however, due to the *group* ⇒ *group* concept-mapping underlying the whole-string bridge between ***abc*** and ***kkjjii***, which conflicts with the *letter* ⇒ *group* slippages underlying the other vertical bridges. In the case of the top themes, a weakly-activated *Alphabetic-Position: identity* theme has appeared, on account of a *first* ⇒ *first* concept-mapping that got noticed and added to the ***a***–***a*** bridge at some point, but this theme is not strong enough to attain dominance. On the other hand, both *Letter-Category* top themes are strongly activated, but neither one is dominant—probably due to the relatively high strength of the ***c***–***d*** bridge, which prevents the *successor* theme from fading away or being suppressed by the competing *identity* theme.

The themes in this example all exhibit positive levels of activation, representing varying estimates of the importance or centrality of particular ideas. However, negative activation levels are also possible for themes, as was mentioned earlier (although negative themes normally arise only under certain special circumstances to be discussed fully in section 4.5.2). Unlike positively-activated themes, a negatively-activated theme represents evidence for the *absence* or *inappropriateness*

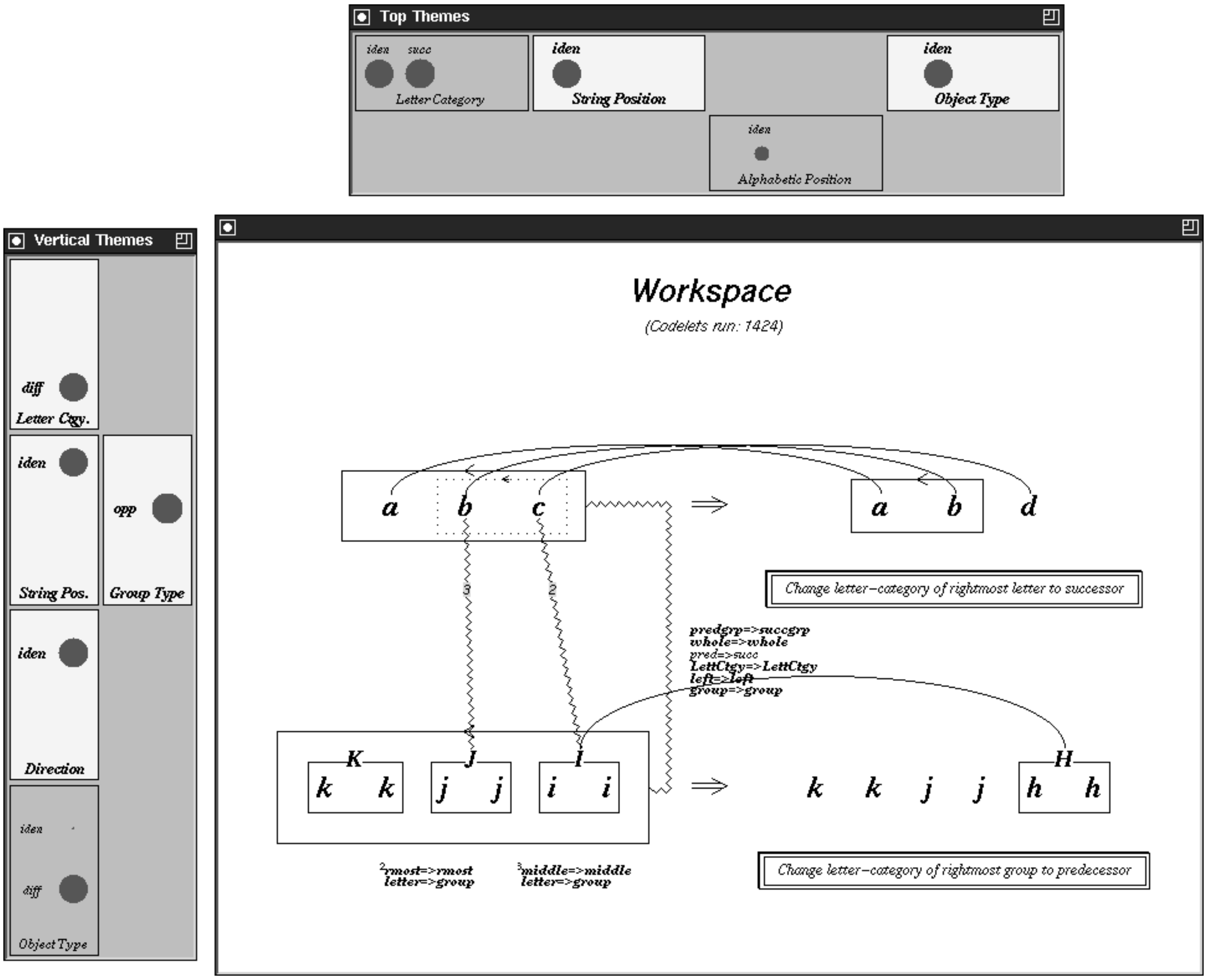


Figure 4.2: *The state of Metacat's Themespace during a run of the problem "**abc** ⇒ **abd**; **kkjjii** ⇒ **?**" after the program has found the answer **kkjjhh**.*

of a particular idea in some situation. For example, a negatively-activated *Letter-Category: identity* top theme represents the notion that letter-category sameness is *not* a key idea underpinning the mapping between the initial string and the modified string.

Accordingly, positive and negative themes within a single theme cluster exhibit different dynamics. Unlike a positive theme, which exerts an inhibitory effect on other positively-activated themes in the same cluster, a negative theme exerts an *excitatory* effect on positive themes within the same cluster. The rationale is that since the negative theme represents the inappropriateness of viewing a mapping in some way, and the positive themes represent some degree of direct evidence for alternative interpretations (with respect to the same theme category), the alternative themes should become more highly activated. For example, a negatively-activated *Letter-Category: identity* theme and a positively-activated *Letter-Category: successor* theme both, in a sense, "pull in the same direction"—that is, away from the idea of letter-category sameness and towards the idea of letter-category successorship—and this tends to reinforce the latter theme.

Likewise, as direct, positive evidence accumulates in support of ideas (*i.e.*, as positive themes become more strongly activated), the relevance of negatively-activated themes diminishes. Accordingly, positive themes exert an *inhibitory* effect on their negative rivals. For example, the building of a new ***c*–*d*** bridge adds a bit of "evidence" in support of the idea of letter-category successorship, thus boosting the activation of the *Letter-Category: successor* theme, which in turn inhibits the negatively-activated *Letter-Category: identity* theme. The overall effect is that a gradual shift away from a negative characterization of a situation (*e.g.*, "letter-category sameness is *not* the key here") to a positive characterization (*e.g.*, "letter-category successorship *is* the key") takes place.

Lastly, in the case of two negatively-activated themes within the same cluster,

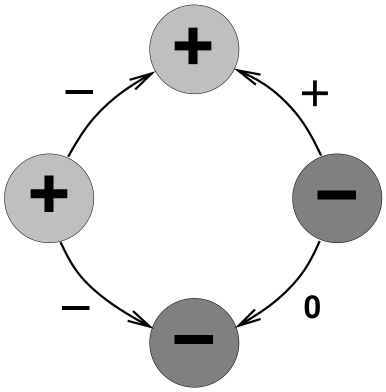


Figure 4.3: *The mutual excitatory and inhibitory effects of themes within a cluster. Positively-activated themes inhibit one another, as well as negatively-activated themes. Negatively-activated themes excite positively-activated themes, but have no effect on one another.*

neither theme has any effect on the other, the rationale being that the inappropriateness of one idea neither supports nor contradicts the inappropriateness of alternative ideas. For example, knowing that letter-category sameness isn't the key does not warrant the conclusion, in the absence of any other positive evidence, that letter-category successorship may well be. These relationships are shown schematically in Figure 4.3.

4.1.2 Top-down influence of themes

In the early stages of development of the Themespace, a number of different approaches were tried for integrating top-down pressure from themes with Workspace processing and other top-down forces in Metacat. Most of these approaches, however, proved to be unsatisfactory.

One such approach involved letting the amount of top-down pressure exerted by a theme vary continuously in proportion to the theme's activation, so that all themes in the Themespace—even if only slightly activated—would continually influence codelet activity in the Workspace. A similar but more indirect approach involved themes spreading activation to their constituent Slipnet concepts—again, continuously as a function of their own activation (a combination of these two approaches was also tried). In addition, the amount of mutual inhibition and excitation exerted by themes on each other within theme clusters was varied, in the hope of finding a good balance that would enable the kind of "locking-in" effect described earlier to occur as a consequence of the mutual interaction between themes and Workspace activity (rather than only from interactions between themes). Another approach involved letting only *dominant* themes influence processing, but this did not work very well either.

In all of these cases, the top-down pressure exerted by themes seemed to hinder Metacat's progress more often than it helped. The reason for this is that such pressure, being automatically applied whenever themes attained sufficient activation, often "distracted" the program by causing it to focus too soon on specific ideas—and usually on too many specific ideas at one time. Instead, an additional mechanism was needed that would allow top-down thematic pressure to be selectively turned on or off, according to the situation at hand. Themespace activations by themselves are not sufficient for achieving the kind of focused and directed behavior that Metacat needs in order for it to gain insight into what it is doing.

Most of the time, therefore, themes behave as passive representational structures, influenced by activity occurring in the Workspace (and by the activations of other themes), but having no return effect on this activity. However, under certain circumstances (which will be explained more fully in subsequent sections), thematic pressure can be turned on by the program itself, strongly influencing subcognitive processing activity in the Workspace, as outlined earlier in Chapter 2. When thematic pressure is turned on, positively-activated themes encourage the building of Workspace structures that are compatible with the ideas represented by the themes. Negatively-activated themes, on the other hand, *discourage* the building of such structures; instead, they promote the building of structures that are *incompatible* with themselves. A negatively-activated *Object-Type: identity* theme, for instance, would promote the creation of bridges between different types of objects (*i.e.*, between a letter and a group), rather than between objects of the same type, such as two letters or two groups.

Top-down thematic pressure in Metacat is realized in three ways. The first (and least focused) way is that whenever thematic pressure is turned on, themes spread activation to their constituent Slipnet concepts (*i.e.*, to the category and relation nodes making up the themes), which may in turn cause top-down codelets to be added to the Coderack, as in Copycat.

Secondly, themes directly influence the strengths of Workspace structures in a dynamic, continuous fashion, as a function of the *thematic compatibility* between themes and structures. (In the current version of Metacat, only the strengths of bridges and descriptions can be influenced by themes, although in principle other types of structures, such as bonds or groups, could also be affected.) A structure's thematic compatibility is determined by how well the structure "resonates" with the ideas represented by the set of currently-active themes in the Themespace. For example, in the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*, a vertical bridge between ***a*** and ***z*** would be incompatible with a vertical *String-Position: identity* theme, due to the *leftmost ⇒ rightmost* slippage underlying the bridge. On the other hand, the vertical themes *String-Position: opposite* and *String-Position: different* would both support the bridge, as would the theme *Alphabetic-Position: opposite*. The equivalent *top* themes, however, would all be indifferent to the bridge (since it is not a top bridge). In general, the thematic compatibility of a bridge with respect to a particular set of themes depends on the concept-mappings underlying the bridge. In the case of descriptions, thematic compatibility is determined by the presence or absence of themes of the same category as the description. For example, *String-Position* descriptions are compatible with *String-Position* themes, but are indifferent to *Alphabetic-Position* themes.

A positively-activated theme that is incompatible with some particular Workspace structure dynamically reduces the strength of the structure in proportion to the theme's level of activation. If the theme supports the structure, the structure's strength is enhanced. If the theme is indifferent to the structure, the strength is not affected, regardless of the theme's level of activation. Conversely, a negatively-activated theme that is incompatible with a structure *enhances* its strength—the idea being that since the negative theme represents the absence or inappropriateness of some idea, structures incompatible with that idea should be promoted, as described earlier. Likewise, a theme that is compatible with a structure *decreases* its strength when negatively activated. And, like positively-activated themes, negatively-activated themes have no effect on structures to which they are indifferent.

Thus, themes act like a set of "knobs" that can be used to smoothly vary the strengths of Workspace structures, "skewing" them away from their "resting strengths" (*i.e.*, the strengths the structures would normally have in the absence of thematic pressure) according to the structures' compatibility with the set of currently-active themes in the Themespace. Through the "twisting of knobs" (*i.e.*, varying the pattern of Themespace activations), Metacat's subcognitive perceptual processing can be steered in different directions, guided by the ideas explicitly represented by the themes.

As a simple example, suppose that in the problem *"**abc** ⇒ **abd**; **iijjkk** ⇒ **?**"*, a bridge with a resting strength of 60 exists between the ***a*** in ***abc*** and the ***ii*** group in ***iijjkk***. Figure 4.4 shows the way in which the bridge's strength would be affected in

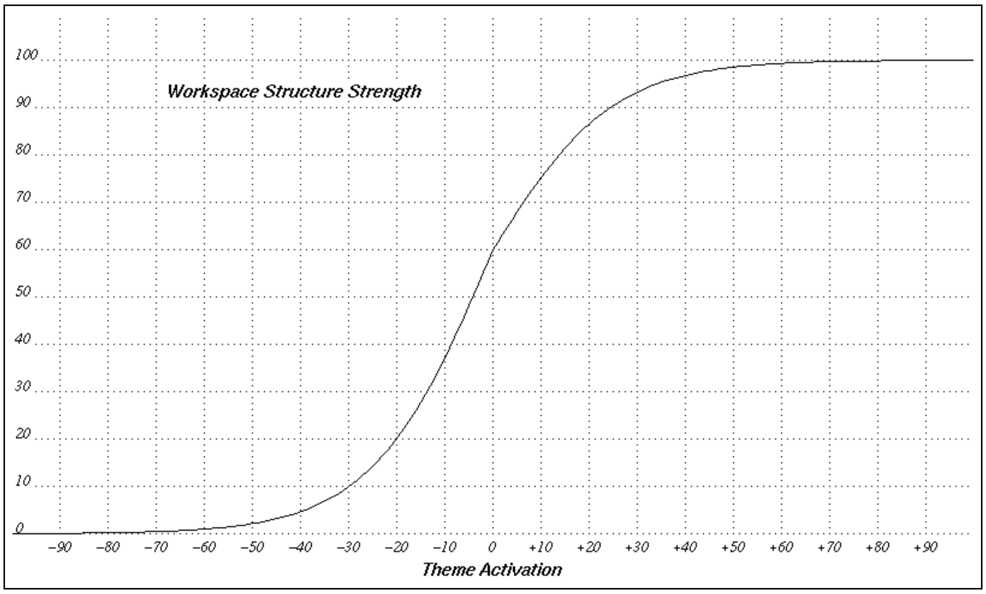


Figure 4.4: *The effect of theme activation on Workspace structure strength. The curve shows the influence of a single theme on a structure of strength 60. (The shape of the response curve differs slightly for negative and positive theme activations.)*

the presence of a single *String-Position: identity* vertical theme with thematic pressure turned on. This particular theme supports the bridge (on account of the bridge's *leftmost* ⇒ *leftmost* concept-mapping), so positively activating the theme causes a corresponding increase in the bridge's strength. Likewise, negatively activating the theme drives the bridge's strength toward zero, since in this case the theme represents the *inappropriateness* of seeing objects as corresponding on the basis of identical string positions. As can be seen from the graph, a relatively small amount of negative theme activation quickly undermines the bridge's strength, while a small amount of positive theme activation quickly boosts it to near-maximum strength. In the absence of any theme activation, the bridge's strength reverts back to its original value of 60.

This example illustrates the effect of a single theme on structure strength. In general, however, many themes may be active at any given moment, some of which may be compatible with a given structure, and some of which may not be. For example, the vertical themes *String-Position: identity*, *Letter-Category: different*, and *Object-Type: identity* might all be active in the above example. Assuming positive theme activations, *String-Position: identity* and *Letter-Category: different* would both support the ***a***–***ii*** bridge, while *Object-Type: identity* would be incompatible with it. In such a case of mixed thematic compatibility, the incompatible themes will tend to "drown out" the compatible themes, even if the latter outnumber the former. The ***a***–***ii*** bridge will therefore remain very weak because of the active and incompatible *Object-Type: identity* theme, in spite of the other two compatible themes. This reflects the idea that only interpretations consistent with *all* highly-activated themes at any given moment should be pursued.

The third way in which themes exert top-down pressure is through *Thematic-bridge-scout* codelets. When thematic pressure is turned on, these codelets explicitly seek to propose bridges that would be compatible with positively-activated themes in the Themespace. For example, if the top themes *Letter-Category: identity* and *String-Position: different* are both active, thematic-scout codelets will tend to look for potential bridges between objects in the initial string and the modified string having the same letter-category but different string positions. On average, the more strongly activated a theme is, the more urgently thematic-scout codelets will tend to look for structures compatible with the theme. Thus, in the problem *"**abc** ⇒ **cab**; **ijk** ⇒ **?**"*, with the above two themes strongly activated, the top bridges ***a***–***a***, ***b***–***b***, and ***c***–***c*** will tend to quickly get proposed by thematic-scout codelets (and subsequently built, since the themes will also strongly boost the strengths of the bridges in the manner described earlier, thereby increasing the likelihood that *Bridge-evaluator* codelets will decide in favor of building the bridges). In the absence of thematic pressure, these bridges are much less likely to be built.

In contrast to positively-activated themes, negatively-activated themes do not exert pressure via thematic-scout codelets—which pay attention only to positively-activated themes in the Themespace. Rather, negatively-activated themes exert pressure only by strengthening or weakening Workspace structures on the basis of thematic compatibility, as described earlier. The reason for this is to prevent thematic-scout codelets from proposing large numbers of spurious bridges on the basis of negative themes. For example, with a negative *Letter-Category: successor* top theme strongly activated, codelets would end up proposing bridges between any two objects that were *not* related by letter-category successorship, which would result in a tangle of proposed bridges between the initial string and modified string, most of which would contribute little or nothing to the emergence of a coherent mapping between the strings.

Finally, *Thematic-bridge-scout* codelets can also encourage new descriptions of existing Workspace objects to be built, if such descriptions would subsequently enable bridges consistent with positively-activated themes to be proposed. For example, in the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*, suppose that the special alphabetic-position status of the letters ***a*** and ***z*** has not yet been noticed—that is, *Alphabetic-Position* descriptions have not yet been attached to ***a*** or ***z***. If a vertical *Alphabetic-Position: opposite* theme becomes active in the presence of thematic pressure, thematic-scout codelets will be on the lookout for objects in the initial string or target string that can be described in terms of their alphabetic-position. If they happen to focus on an object that lacks such a description, they will try to propose one for the object, if possible. Thus the active *Alphabetic-Position: opposite* theme encourages alphabetic-position descriptions to be attached to ***a*** and ***z***, which in turn paves the way for the creation of an ***a***–***z*** bridge supported by the slippage *first* ⇒ *last*. This reflects the idea that situations tend to be perceived in terms of the features that one is actively paying attention to.

In a similar fashion, new relationships between objects linked by a bridge created under thematic pressure can sometimes be noticed "in retrospect". Returning to the previous example, if an ***a***–***z*** bridge is created under pressure from a vertical *String-Position: opposite* theme, supported by the slippage *leftmost* ⇒ *rightmost* (but not by *first* ⇒ *last*, due to the absence of an active *Alphabetic-Position: opposite* theme), there is still some chance that the alphabetic-position symmetry between ***a*** and ***z*** will be noticed as a result of the new bridge, on account of the symmetric *leftmost* ⇒ *rightmost* slippage having been made under thematic pressure. The active *String-Position: opposite* theme spreads activation to the *opposite* concept in the Slipnet, which in turn makes other slippages based on this concept more likely. Furthermore, the concepts *leftmost* and *rightmost* are linked to the concepts of *first* and *last* in the Slipnet. Thus a thematic-scout codelet may notice the "parallel" *first* ⇒ *last* relationship between ***a*** and ***z***, in which case the new slippage would then be added to the bridge's concept-mappings. This idea is related to the notion of "coattail slippages", discussed in Chapter 3.

4.2 Patterns

The foregoing discussion explains the mechanisms through which top-down thematic pressure is applied in Metacat, but does not address the circumstances under which this occurs. However, a few examples illustrating the application of thematic pressure have already been outlined in sections 2.4.4 and 2.4.5 of Chapter 2. As will be recalled, one situation in which top-down thematic pressure may be turned on arises when Metacat has "fallen into a rut" by trying to apply the same set of ideas to a problem over and over, without success. Repeatedly hitting the same snag may eventually cause Metacat to clamp the themes characterizing the snag with strong negative activation, which in turn may nudge the program out of its unproductive cycle of behavior, toward alternative ways of looking at the problem. Another such situation arises when Metacat runs in justify mode, attempting to make sense of an answer provided to it. As a result of comparing different rules for describing string changes, the program may clamp a particular set of themes with strong positive activation, in an attempt to discover a globally consistent interpretation of the given problem that leads to the given answer. In both of these cases, the clamping of theme activations in the Themespace automatically turns on thematic pressure, and can be thought of as Metacat's way of explicitly focusing on a particular set of ideas in order to explore their implications.

The ability of the program to clamp theme activations, however, is actually just one manifestation of a more general ability, in which various types of *patterns* can be clamped by the program in response to different situations. In addition to patterns of theme activations, patterns of concept activations in the Slipnet and patterns of codelet urgencies in the Coderack can also be clamped, depending on the circumstances.

4.2.1 Theme-patterns

In general, a *theme-pattern* may consist of any number of themes of a particular type, along with an optional positive or negative activation value for each theme (in the absence of a specific value, full positive activation is assumed). For example, a possible vertical theme-pattern is shown below:

```
(vertical-themes
   (String-Position: opposite 100)
   (Direction: opposite 100)
   (Group-Type: identity 100))
```

Clamping this theme-pattern automatically turns on thematic pressure, strongly encouraging the creation of a "crosswise" mapping between the initial string and target string—especially between groups of the same type within these strings. For example, Figure 4.5 shows the Workspace and Themespace after Metacat found the

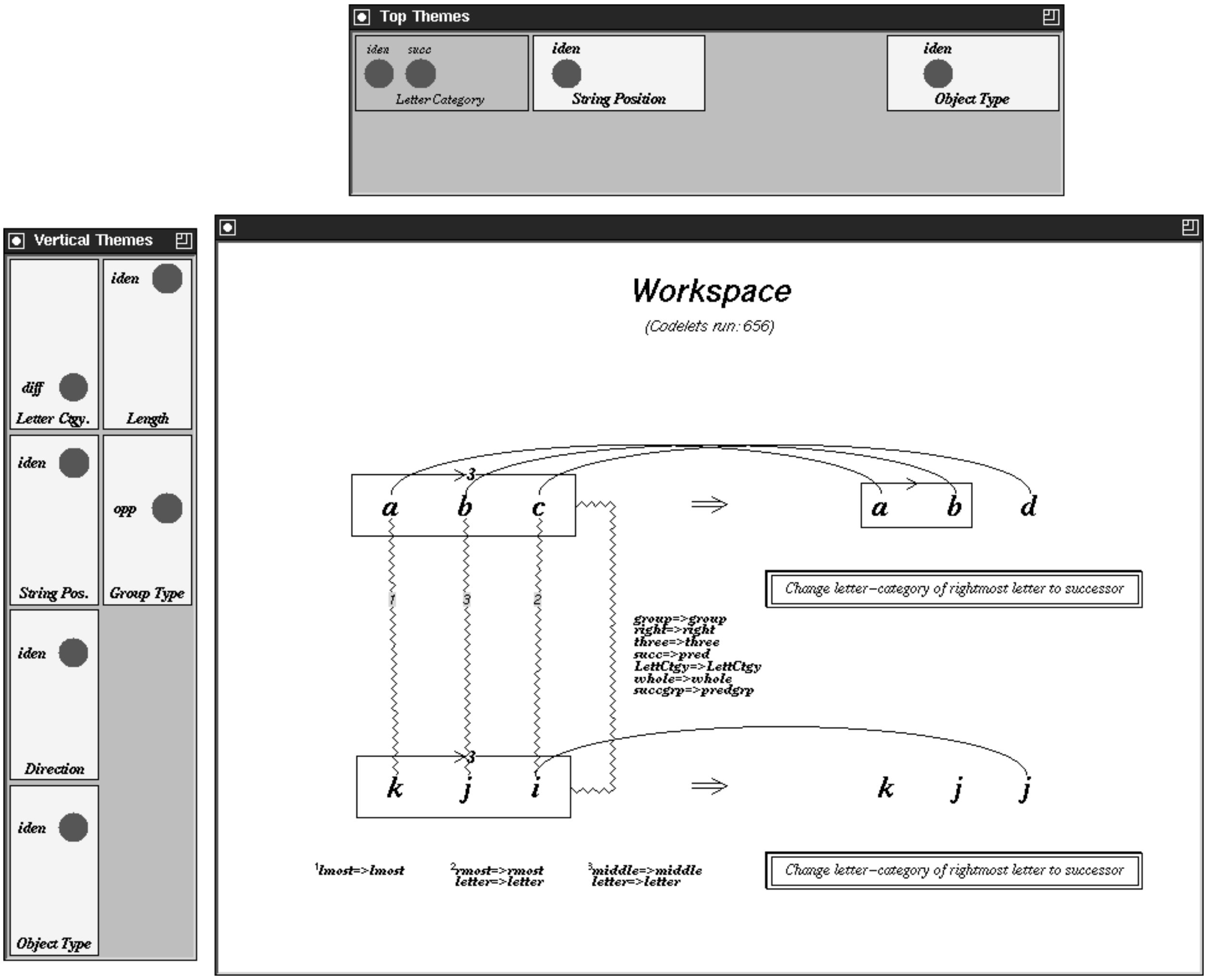


Figure 4.5: *The answer* ***kjj*** *to the problem "****abc*** ⇒ ***abd****;* ***kji*** ⇒ ***?****", showing the state of the Themespace at the time the answer was found.*

answer ***kjj*** to the problem *"**abc** ⇒ **abd**; **kji** ⇒ ?"* (in the absence of thematic pressure). In this particular run, Metacat saw ***abc*** and ***kji*** as groups of different types going in the same direction. To demonstrate the effect of thematic pressure, the above theme-pattern was then *manually* clamped (by me), along with a top theme-pattern consisting of the themes *String-Position: identity* and *Object-Type: identity*, and the run was continued. Under the influence of the clamped patterns, Metacat quickly reorganized its way of looking at ***kji*** (but maintained the same interpretation of ***abc*** ⇒ ***abd***), resulting in the answer ***lji*** approximately 100 codelets later (see Figure 4.6). Normally, however, Metacat itself is responsible for clamping and unclamping patterns of themes as it sees fit, rather than relying on a human to tell it specifically which ideas to "think about".

Theme-patterns are associated with various types of structures and processing events in Metacat, including snags (as was mentioned earlier), answer descriptions, slippages, and rules. These associations will be spelled out more clearly later in this chapter (and illustrated in sample runs of the program in the next chapter), but to take just one example here, consider the rule *Change letter-category of rightmost letter to successor* from the earlier run shown in Figure 4.5. This rule is based on seeing ***abc*** and ***abd*** as going in the same direction, represented by the horizontal bridges ***a***–***a***, ***b***–***b***, and ***c***–***d***. When this rule gets created, the two top themes *String-Position: identity* and *Object-Type: identity* are dominant, since they are supported by all three bridges. This pattern of dominant themes can be thought of as abstractly characterizing the "background of similarity" that exists between ***abc*** and ***abd***, against which the differences between the strings are perceived.[3] Consequently, this top theme-pattern (shown clamped in the upper window of Figure 4.6) is permanently associated with the newly-created rule.

[3]The idea of a "background of similarity" between strings and its relationship to rule creation was discussed in section 3.1 of Chapter 3.

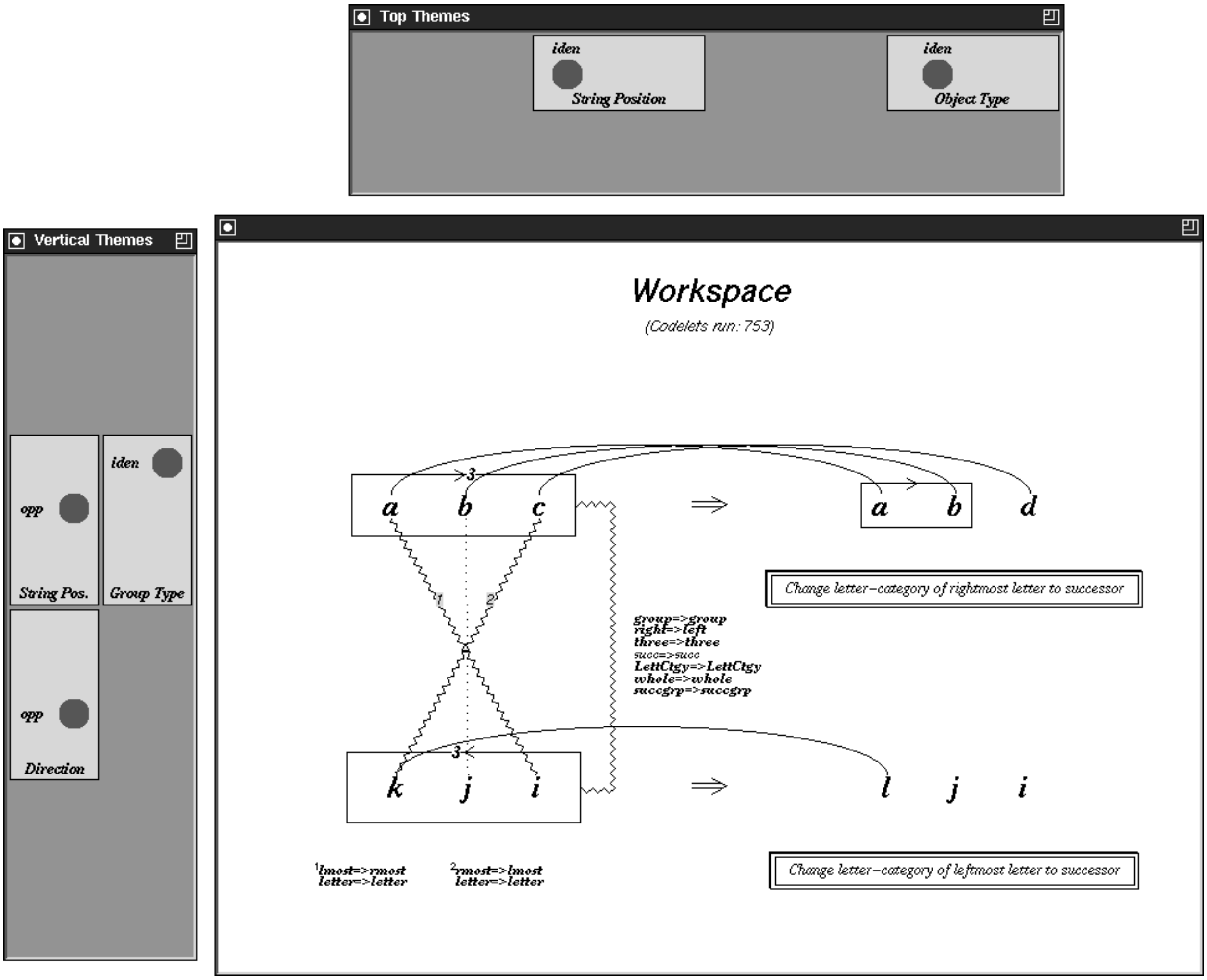


Figure 4.6: *The answer* ***lji****, found as the result of manually clamping the theme-patterns shown. This run is a continuation of the run from Figure 4.5.*

4.2.2 Concept-patterns

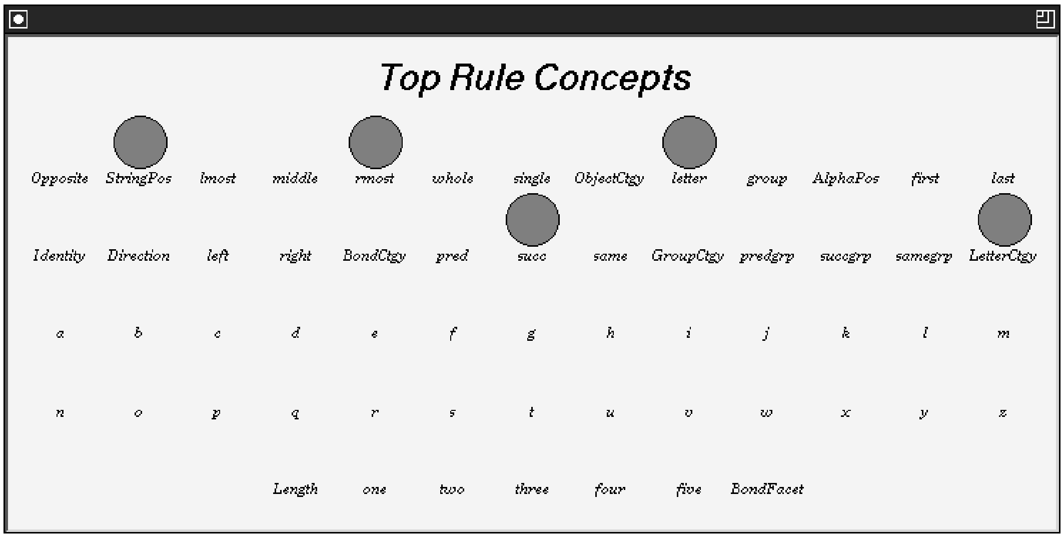
A *concept-pattern* is similar to a theme-pattern, except that it specifies a set of Slipnet concepts (along with an activation value for each concept) instead of a set of themes. Like theme-patterns, concept-patterns can be clamped by the program in various situations, and are associated with several types of structures, including theme-patterns, rules, slippages, and Workspace objects. The concept-pattern below, for example, is associated with the vertical theme-pattern shown earlier:

```
(concepts
   (String-Position 100)
   (Direction 100)
   (Group-Category 100)
   (opposite 100))
```

Clamping a theme-pattern causes its associated concept-pattern to be clamped in the Slipnet, further strengthening the effects of top-down pressure. Other examples of concept-patterns are displayed graphically in Figure 4.7. The upper pattern in the figure is associated with the rule *Change letter-category of rightmost letter to successor*, and consists of the concepts making up the rule. The lower pattern is associated with the right-directed predecessor group ***kji*** created in the example discussed earlier, and consists of the concepts making up the group's descriptions. (The pattern specifies an activation of zero for the concept of *three* because the group's length description happened not to be relevant at the time the group was created.)

4.2.3 Codelet-patterns

In addition to patterns of themes and concepts, Metacat can also clamp *codelet-patterns*, which specify sets of codelet types and urgency values. Clamping a codelet-pattern dynamically alters the urgencies of codelets in the Coderack according to their type, effectively overriding the urgencies that were individually assigned to codelets at the time of their creation. Furthermore, any new codelets added to the Coderack



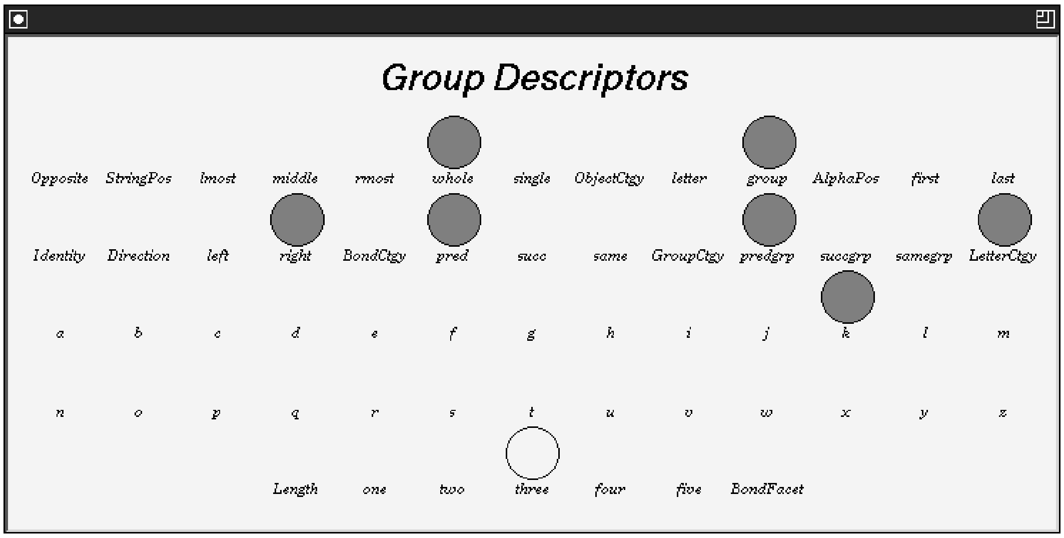


Figure 4.7: *Two examples of concept-patterns. The upper pattern is associated with the top rule shown in Figure 4.5, while the lower pattern is associated with the* ***kji*** *predecessor group.*

automatically assume the urgencies specified by the codelet-pattern, for as long as the pattern remains clamped.

For example, Figure 4.8 shows the effect of clamping a pattern that specifies high urgencies for *Bottom-up-bridge-scout*, *Important-object-bridge-scout*, *Bridge-evaluator*, *Bridge-builder*, *Rule-scout*, *Rule-evaluator*, and *Rule-builder* codelets, and low urgencies for all other types of codelets. (In general, the urgency levels of a clamped codelet-pattern are indicated by shades of grey in the Coderack.) The left image shows the probabilities for selecting codelets of each type from the Coderack before clamping the pattern. The actual frequency of each type of codelet in the current codelet population is also shown. As can be seen, clamping the codelet-pattern significantly alters the selection probabilities of codelets (although not their frequencies). This dramatically speeds up the search for new bridges and rules. Thus, codelet-patterns can be used to effectively "channel" the parallel terraced scan in very specific directions.

4.3 Answer Justification

Metacat's ability to clamp various types of patterns plays a crucial role in its ability to make sense of answers provided to it when running "backwards" in justify mode. This process, outlined earlier in section 2.4.5, will now be described in detail. (Several complete sample runs of the program illustrating the ideas described here will be presented in the next chapter.)

Justifying a given answer involves essentially the same mechanisms used by Metacat when searching for answers on its own, except that four strings exist instead of just three—the extra one being the answer string provided by the user. Accordingly, all four strings are examined by codelets looking for potential structures to build (*i.e.*, bonds, groups, or descriptions). Furthermore, a third string mapping is created (the "bottom mapping"), consisting of horizontal bridges between the target string and

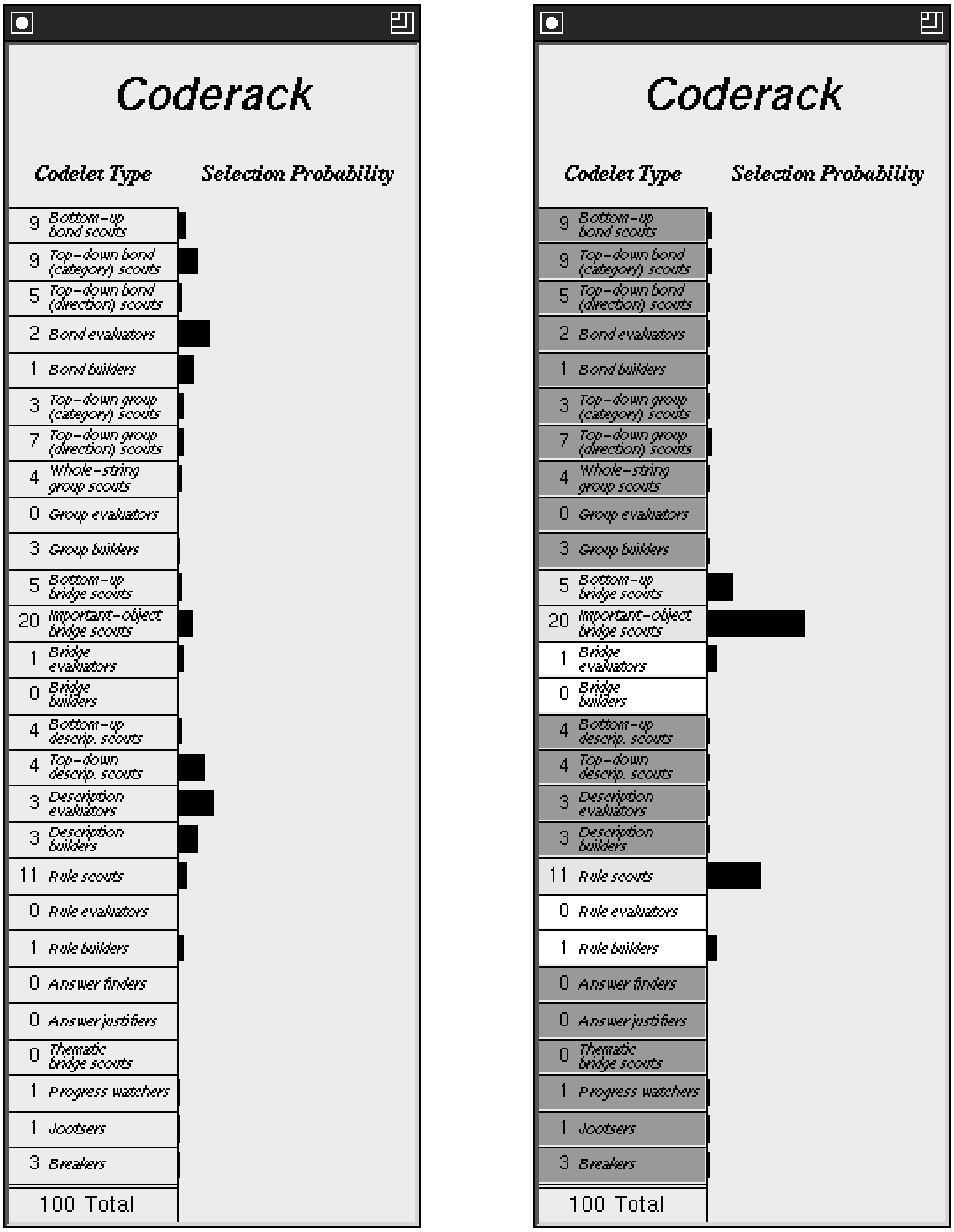


Figure 4.8: *The effect of clamping a codelet-pattern on the selection probabilities of codelets in Metacat's Coderack. Clamped urgency levels are indicated by shades of grey, with lighter shades corresponding to higher urgencies. Codelet frequencies are also shown. Selection probabilities are a function of both urgencies and frequencies. (The differences between the types of codelets that exist in Metacat and in Copycat can be seen by comparing this figure with Figure 1.5 on page 31.)*

the answer string, as well as "bottom rules" based on this mapping that describe how the target string changes into the answer string.

In order to make sense of an answer, Metacat must discover three things: (1) a way of describing the change from the initial string to the modified string; (2) an *analogous* way of describing the change from the target string to the answer string; and (3) a way of seeing the target string as being similar to the initial string that is consistent with (1) and (2). That is, the objects that change in the target string (as described by the bottom rule) must be seen as playing *the same roles* in the target string that the objects described by the top rule as changing play in the initial string. In other words, vertical bridges linking the target-string objects to their corresponding initial-string objects must exist, based on concept-mappings (perhaps including slippages) between the ideas inherent in the top rule and those inherent in the bottom rule.

For example, suppose that in the problem *"**abc** ⇒ **cba**; **ppqq** ⇒ **qqpp**"*, a top rule describing ***abc*** ⇒ ***cba*** as *Swap positions of leftmost letter and rightmost letter* (based on horizontal bridges ***a***–***a*** and ***c***–***c***), and a bottom rule describing ***ppqq*** ⇒ ***qqpp*** as *Swap positions of leftmost group and rightmost group* (based on bridges ***pp***–***pp*** and ***qq***–***qq***) have been created, along with vertical bridges ***a***–***pp*** and ***c***–***qq*** (supported by *letter* ⇒ *group* slippages). Taken together, these structures serve as an explanation for how the answer ***qqpp*** arises: the leftmost and rightmost letters in the top situation are viewed as swapping their positions; the leftmost and rightmost *groups* in the bottom situation are viewed as the counterparts to the top situation's letters; therefore, doing "the same thing" in the bottom situation amounts to swapping the positions of the groups, yielding ***qqpp***.

4.3.1 Answer-justifier codelets

In Metacat, *Answer-justifier* codelets are responsible for examining structures in the Workspace to see if they meet the above three criteria. Justifier codelets first choose either a top or bottom rule (as a function of rule strength) and attempt to translate the chosen rule based on the vertical mapping that exists between the initial string and target string. If the resulting translated rule matches some existing rule, and if this rule and the chosen rule are both currently supported,[4] then a consistent way of interpreting both the top and bottom string changes has been found that yields the answer string. The program therefore pauses to report the new interpretation, displaying the top and bottom rules along with the accompanying bridges and concept-mappings.

For instance, in the previous example, a justifier codelet might choose the bottom rule *Swap positions of leftmost group and rightmost group* for translation. In the presence of *letter* ⇒ *group* slippages underlying the vertical bridges, this rule translates to *Swap positions of leftmost letter and rightmost letter*, which matches the existing top rule, indicating that a consistent interpretation has been found. (As this example shows, rule translation can occur in either direction when Metacat runs in justify mode—from top rules to bottom rules, or from bottom rules to top rules.) On the other hand, if the letter ***a*** in ***abc*** happened to correspond to the leftmost *letter* in ***ppqq***, instead of to the leftmost group (or if ***c*** were seen as corresponding to the rightmost letter ***q***), then the resulting translated rule would not match the top rule, since both *letter* ⇒ *group* slippages are necessary for the translation to yield a matching rule, and thus no answer justification could occur.

Another possibility is that the translated version of a rule may not match any existing rule but may in fact work correctly (*i.e.*, it may correctly describe the way in which a string changes). For instance, suppose that only the top rule in the previous example exists at the time the justifier codelet runs. The codelet would thus choose the top rule, translating it as *Swap positions of leftmost group and rightmost*

[4]As was explained in section 3.4 of Chapter 3, a rule is supported if all of the horizontal bridges on which the rule was originally based currently exist in the Workspace.

group (assuming the existence of the vertical *letter* ⇒ *group* bridges). Although the translated rule in this case does not match any bottom rule (since no bottom rules have yet been built), it nevertheless describes the ***ppqq*** ⇒ ***qqpp*** change correctly, and is therefore added to the Workspace as a new bottom rule. Together with the top rule and vertical bridges, this new rule provides a coherent justification for the answer ***qqpp***.[5]

Justifying answers via top-down pressure

In a sense, the foregoing examples represent the easiest cases of answer justification, because all of the Workspace structures needed to justify an answer exist (or can be easily created by translating a rule, as in the last example) at the time the *Answer-justifier* codelet runs. All the codelet needs to do is recognize that, based on the existing vertical mapping, a pair of existing rules will in fact produce the given answer string. A more interesting situation arises, however, if the current configuration of Workspace structures *almost* provides a consistent interpretation of the answer, but falls short in some way. This inconsistency may suggest a way of reorganizing structures under top-down pressure, so that a consistent interpretation can be achieved. Depending on the type of inconsistency detected, a justifier codelet may clamp various types of patterns in order to force this perceptual reorganization to occur.

There are several different types of inconsistent situations that can trigger the clamping of patterns by justifier codelets. In the first case, one of the rules may be unsupported. In other words, the horizontal bridges that exist at the time the justifier codelet runs may no longer accurately reflect the relationship between the strings as described by the rule. For example, Figure 4.9 shows the problem discussed earlier,

[5]This example assumes that the newly-created rule is supported—in this case by bridges ***pp***–***pp*** and ***qq***–***qq***. The next section explains what happens if this is not the case.

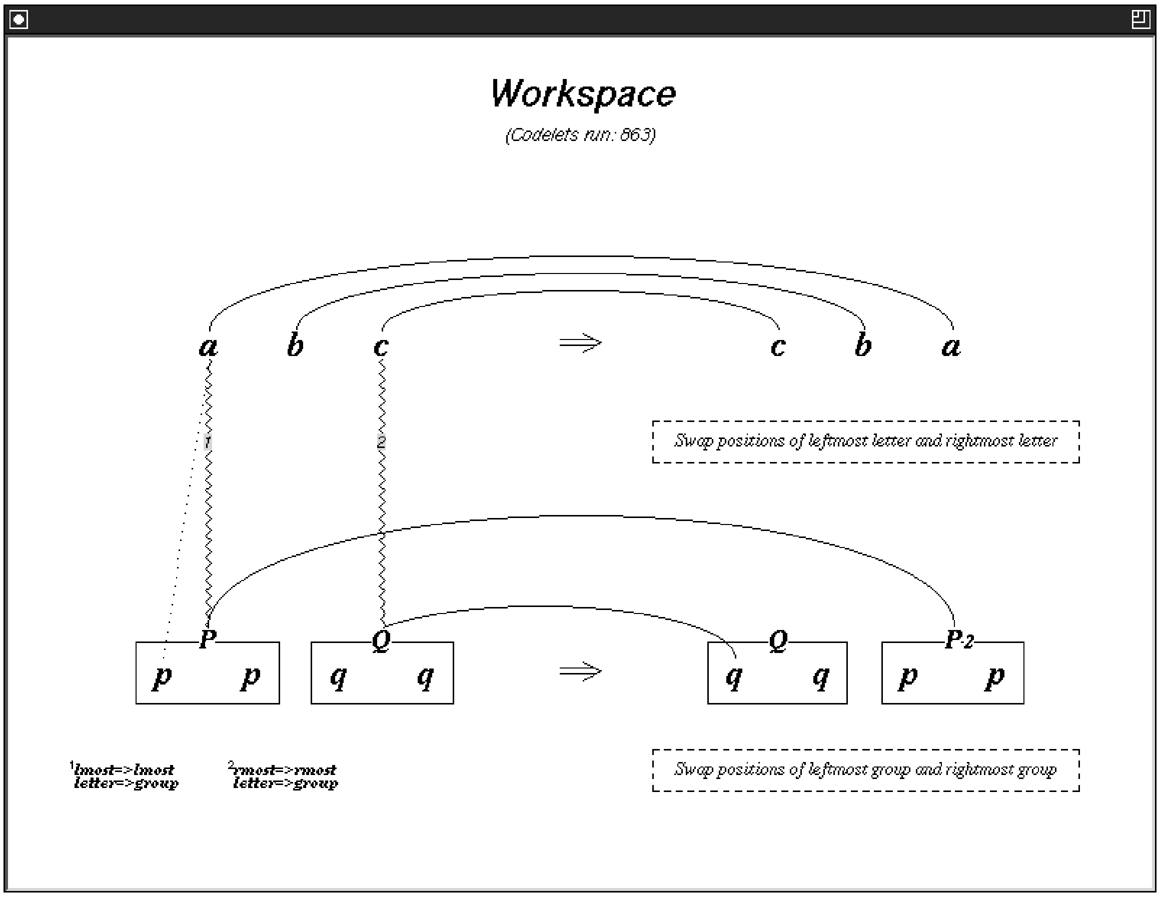


Figure 4.9: *An example of an unsupported bottom rule.*

in which both the top and bottom strings are viewed as swapping their leftmost and rightmost objects (letters in one case and groups in the other). The vertical bridges support this interpretation, since they map letters to groups; likewise, the top horizontal bridges are consistent with the notion of the letters ***a*** and ***c*** swapping positions. However, the bottom ***qq***–***q*** bridge is inconsistent with the bottom rule. To be consistent (*i.e.*, for the rule to be supported), the two ***qq*** groups must be seen as corresponding to one another. A similar type of inconsistency may arise if a justifier codelet creates from scratch a new rule that works correctly (by translating some existing rule, as described at the end of the previous section), but which happens not

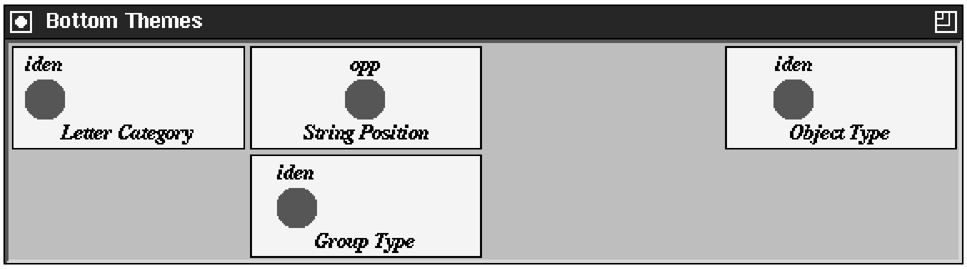


Figure 4.10: *The theme-pattern associated with the bottom rule of Figure 4.9.*

to be supported by the appropriate horizontal bridges.

In both of these cases, the justifier codelet responds to the inconsistent situation by clamping a set of patterns that focus top-down pressure on the problematic horizontal mapping, encouraging its reorganization in a manner consistent with the unsupported rule. Equally important, these patterns also exert pressure to *preserve* those structures that already form part of a consistent interpretation of the answer. Specifically, the following patterns get clamped:

- *The theme-patterns associated with the top and bottom rules.* The effect of clamping a rule's theme-pattern depends on whether or not the rule is supported. If it is, then the pressure exerted by the themes will tend to hold the current set of bridges in place, since the concept-mappings underlying these bridges are compatible with the clamped themes. However, if the rule is not supported, the clamped themes will weaken the incompatible bridges, encouraging the creation of new bridges supporting the rule. For example, the theme-pattern associated with the bottom rule in Figure 4.9 is shown in Figure 4.10 (the top rule's theme-pattern is similar, except that it lacks a *Group-Type* theme, since the top mapping involves only letters). Under pressure from these themes—the *Object-Type: identity* theme in particular—the ***qq***–***q*** bridge is likely to be replaced by a ***qq***–***qq*** bridge, while the existing ***pp***–***pp*** bridge is likely to be preserved, resulting in a mapping consistent with the bottom rule. In contrast, the top mapping already supports the top rule, so clamping this rule's theme-pattern just reinforces the existing bridges, locking in the current interpretation of ***abc*** ⇒ ***cba***.

- *The current dominant vertical theme-pattern.* Clamping this pattern reinforces the existing vertical bridges, so that the concept-mappings relating the top rule to the bottom rule will be preserved while the inconsistent horizontal mapping is reorganized. Since the dominant vertical themes represent the ideas that are most likely to be of central importance to the vertical mapping, clamping these themes locks in the current way in which the initial and target strings are viewed as being similar.

- *The unsupported rule's concept-pattern.* The pattern of Slipnet concepts making up the unsupported rule also gets clamped, in addition to the rule's theme-pattern. The top-down pressure exerted by these concepts (through spreading activation and the spawning of top-down codelets, as in Copycat) "complements", in some sense, the pressure exerted by themes, by promoting the creation of other types of structures besides bridges that may be useful in reorganizing the inconsistent mapping. For example, the concept-pattern associated with the bottom rule in Figure 4.9 consists of the concepts *String-Position*, *leftmost*, *rightmost*, and *group*. If the ***qq*** group in ***qqpp*** did not exist at the time of the clamp, this pattern would encourage its creation, which is necessary in order to create the ***qq***–***qq*** bridge.

- *A codelet-pattern accentuating the urgencies of top-down codelets.* Whenever a justifier codelet clamps a set of theme- and concept-patterns in response to inconsistencies in the interpretation of an answer, it also clamps a special codelet-pattern that heightens the effect of the other clamped patterns by speeding up the processing performed by top-down codelets. In a sense, this serves to further catalyze the reorganization of structures in the Workspace, in order that a consistent interpretation of the answer might be discovered more quickly, before the clamp period expires.[6] Specifically, this codelet-pattern imposes very high urgencies on top-down *Bond-scout*, *Group-scout*, and *Description-scout* codelets, on their associated *Evaluator* and *Builder* codelets, and on *Thematic-bridge-scout* codelets. (Other codelet types are not affected.)

Justifying answers via rule unification

Another type of inconsistency that can trigger pattern-clamping by justifier codelets arises if the vertical mapping does not reflect the relationship between the initial string and target string as expressed by the top and bottom rules. One example of this was outlined earlier in section 2.4.5 of Chapter 2. To take another example, the two rules in Figure 4.9 require that the leftmost and rightmost *letters* of ***abc*** correspond to the leftmost and rightmost *groups* of ***ppqq***. A vertical mapping that does not meet this requirement makes no sense in conjunction with these two rules. However, faced with an inconsistent vertical mapping, a justifier codelet may be able to induce a reorganization of the mapping by clamping a pattern of vertical themes that can be derived by comparing the rules to each other. For instance, comparing the rules in Figure 4.9 implies the need for vertical bridges based on the slippage *letter* ⇒ *group*, an idea that can be captured by a vertical *Object-Type: different* theme. Clamping this theme would promote the creation of vertical bridges between letters and groups, leading to a mapping compatible with the rules in question.[7]

In general, if a justifier codelet is unable to find a pair of top and bottom rules

[6] As might be expected, patterns do not remain clamped forever. How and when Metacat decides to unclamp a set of clamped patterns will be described in section 4.5.1.

[7] In Figure 4.9, of course, the vertical mapping is already consistent with the rules, but suppose that the bridges mapped letters to letters, rather than letters to groups.

that match, under translation, with respect to the concept-mappings underlying the current vertical bridges, it will look for a pair of rules that can *potentially* match with respect to *some* set of concept-mappings. For many rule-pairs, of course, this is not possible. For instance, in the earlier example, the ***abc*** ⇒ ***cba*** change might also be described by the top rule *Reverse direction of string*. This rule cannot be translated to yield the bottom rule *Swap positions of leftmost group and rightmost group* under any circumstances, because the rules are not structurally similar. To be "inter-translatable", a pair of rules must share the same internal structure and differ only by pairs of concepts that are potentially slippable (*i.e.*, the concepts must be linked in the Slipnet). For example, the rules *Change letter-category of rightmost letter to successor* and *Change letter-category of rightmost letter to 'd'*, though structurally similar, are not inter-translatable, because no slippage is possible between the concepts of *successor* and *d*.

The process of analyzing a pair of rules in order to determine if there is a set of slippages that will make the rules match under translation is termed *rule unification*. If a justifier codelet identifies a pair of existing rules that can be unified, it creates a vertical theme-pattern based on the set of unifying slippages, which it then clamps along with the other theme- and concept-patterns associated with the rules. Clamping the vertical theme-pattern encourages the creation of a vertical mapping that supports translating the top rule into the bottom rule, while clamping the other patterns has the effect of either maintaining the existing horizontal mappings if they support the rules, or reorganizing them if they don't (as in the cases described earlier).

4.4 The Temporal Trace

Metacat's ability to revise its perception of a problem by clamping patterns of themes and concepts in response to various situations affords it a very powerful degree of

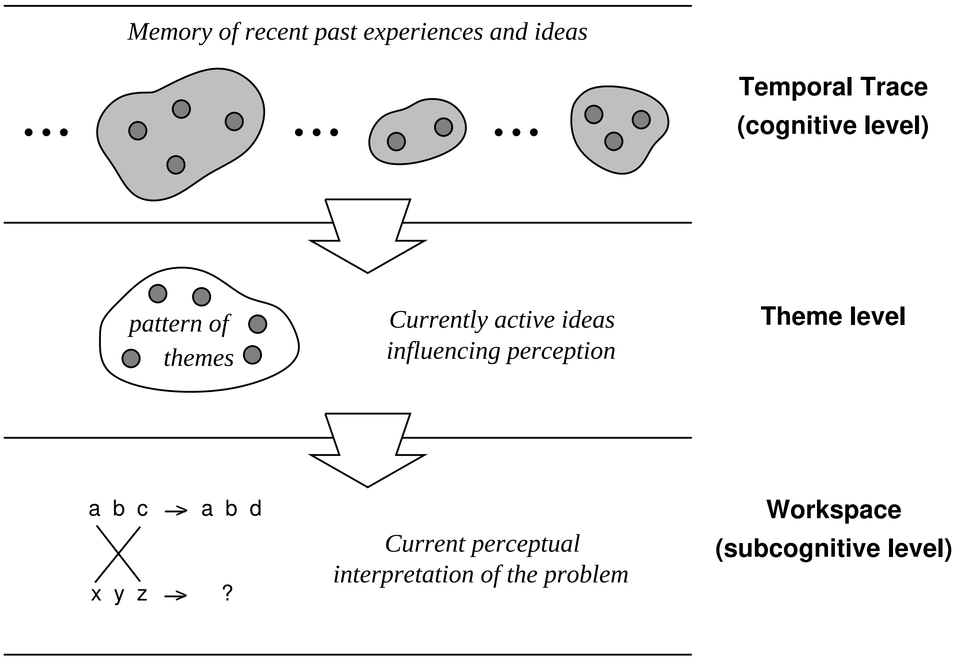


Figure 4.11: *The three levels of processing that exist in Metacat.*

self-control. Patterns—especially patterns of themes in the Themespace—act as a "medium" through which the program is able to wield control over its own behavior, forming a kind of intermediate level sitting above (and strongly influencing) the subcognitive processing level, while remaining below the cognitive level (see Figure 4.11). Metacat's cognitive level is represented by the Temporal Trace, which can be thought of as a short-term memory for storing recent past experiences during a single run of the program. (In contrast, a set of patterns at the intermediate level can be thought of as reflecting the program's *immediate* experience or "state of mind".) Structures and patterns representing several different types of subcognitive-level and intermediate-level processing events (to be described shortly) are stored in the Trace. As was sketched in Chapter 2, codelets can examine these structures—possibly clamping new intermediate-level theme-patterns as a result—allowing the program to "see" what it is doing and to respond accordingly. The cognitive level thus exerts control over the intermediate level, which in turn guides processing at the subcognitive level, establishing a chain of causality that flows from a highly-chunked representation of the program's behavior down to the myriad constituent micro-events out of which that behavior emerges. From a theoretical standpoint, however, the structures stored in the Trace are no different from other types of perceptual structures stored in the Workspace, since they are all subject to processing by codelets, so in an important sense, the Temporal Trace and the Workspace *can be identified with each other*—implying a kind of "level collapse" between Metacat's cognitive and subcognitive levels (see [Hofstadter, 1979], especially Chapter 20, for an extensive discussion of multi-leveled hierarchical systems that twist back on themselves in a similar fashion). Thus, conceptually, the same set of general processing mechanisms is responsible for perception, for *self*-perception (*i.e.*, self-monitoring), and for self-control in Metacat.

One way to appreciate the abstract, chunked nature of the information stored in the Temporal Trace is to consider the number of "steps" that occur during a typical run of Metacat. At a very fine-grained level of description, where each step corresponds to an action performed by a single codelet, a run consists of many hundreds or thousands of steps. At this level of description, no two runs are ever *exactly* the same, even if they involve exactly the same letter-strings.[8] 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 "macroscopic" processing event—each one of which may itself comprise the actions of many codelets.

For example, Figure 4.12 shows the contents of the Trace after a run of the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"*, in which the program, after trying unsuccessfully a couple of times to take the successor of ***z***, answers ***xyd***. The events that occur during the run appear in the Trace in chronological order, from left to right. Although this

[8] Unless, of course, both runs start out with exactly the same random number seed (in addition to the same letter-strings).

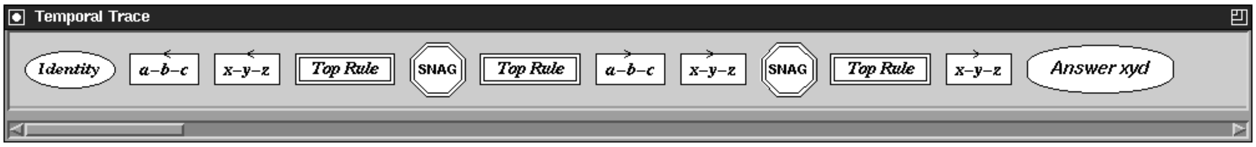


Figure 4.12: *The temporal record of a run of the problem "**abc** ⇒ **abd**; **xyz** ⇒ ?".*

run involves a total of 1,558 codelets, the high-level picture of the run represented 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. Such events include the activation of abstract concepts in the Slipnet; perceiving entire strings as single, chunked wholes; creating new rules for describing string changes; hitting a snag; and discovering a new answer.

For instance, as can be seen from the figure, the concept of *identity* gets activated early on in this particular run (due to the creation of a horizontal ***b***–***b*** bridge between ***abc*** and ***abd***). This is followed by the perception of both ***abc*** and ***xyz*** as whole-string predecessor groups going in the same direction (to the left). The next event records the creation of the rule *Change letter-category of rightmost letter to successor* for describing ***abc*** ⇒ ***abd***, which, given the previous two events, leads inexorably to a snag. In the aftermath of the snag, another rule is created (*Change letter-category of rightmost letter to 'd'*), and ***abc*** and ***xyz*** are reperceived as *successor* groups (again going in the same direction—only this time to the right). However, the program again attempts to use the first rule, resulting in another snag. Finally, after creating a third rule (*Change letter-category of letter 'c' to 'd'*) and again perceiving ***xyz*** as a successor group, the program finds the answer ***xyd***.

As another example, Figure 4.13 shows the history of a run of the same problem in justify mode, in which the answer ***wyz*** has been provided to the program. (This particular run, which involves a total of 953 codelets, was used as the example of

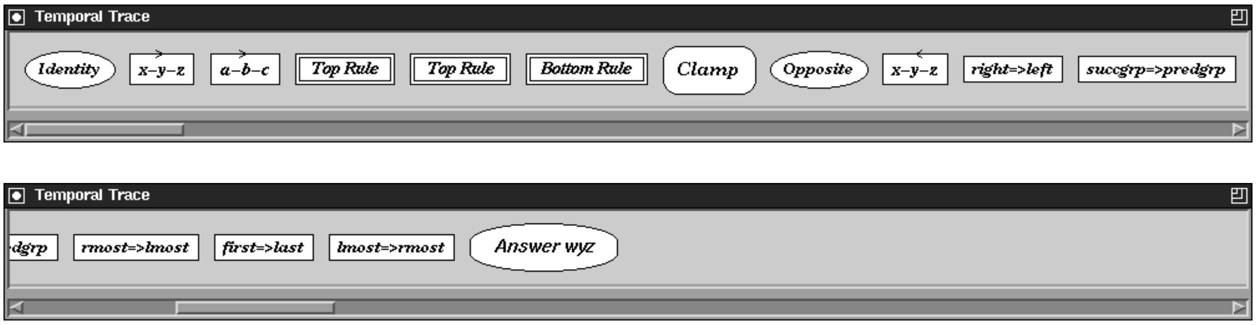


Figure 4.13: *The temporal record of a justification run of the problem "**abc** ⇒ **abd**; **xyz** ⇒ **?**", in which the program was given the answer **wyz**. (The lower image is a continuation of the upper image.)*

"working backwards" discussed in section 2.4.5 of Chapter 2.) As in the previous case, this run's "story line" is clearly discernible from the high-level trail of events in the Trace. After perceiving both ***xyz*** and ***abc*** as successor groups and creating a few rules, the program clamps a set of patterns (as a result of unifying one of the top rules with the bottom rule). This causes the *opposite* concept to become activated, followed by the reinterpretation of ***xyz*** as a predecessor group. The next two events record slippages made when a bridge is created between the right-directed successor group ***abc*** and the left-directed predecessor group ***xyz***. This is followed by three more slippages that subsequently arise from the bridges ***a***–***z*** and ***c***–***x*** (including *first* ⇒ *last*), which leads to a coherent interpretation of the answer ***wyz***.

These two runs illustrate all of the different types of events that can be recorded in the Temporal Trace during a run. Each event, whether occurring in the Workspace, the Themespace, or the Slipnet, has an *importance* value associated with it, and only those events with an importance value above some threshold get explicitly represented in the Trace, allowing Metacat to effectively filter out the "background noise" of a run. Specifically, builder codelets responsible for creating new Workspace structures monitor the importance of the structures they create, adding a new event to the Trace whenever a sufficiently important group, slippage, or rule is built. Likewise, codelets that clamp patterns or attempt unsuccessfully to apply a rule (thus hitting a snag) also note these events in the Trace. In addition, nodes in the Slipnet monitor their own levels of activation, adding new concept-activation events to the Trace whenever sufficiently large changes occur in the activations of deep concepts. Furthermore, all events stored in the Trace, regardless of their type, record the time of occurrence of the event (as measured by the number of codelets run), along with the Workspace structures and Themespace patterns that exist at the time of the event. This auxiliary information can then be referred to later by codelets monitoring the progress of a run.

The list below summarizes the various types of Temporal Trace events that are possible:

- *Concept-activation events* record changes in the activation levels of Slipnet nodes. The importance of this type of event is a function of a node's conceptual depth and of the magnitude of its activation change, with larger changes to deeper concepts being more important.

- *Group events* record the creation of important groups in the Workspace. The importance of this type of event is a function of a group's strength and size, with single-letter groups and whole-string groups being particularly important.

- *Slippage events* record important slippages that occur in support of bridges built between objects in the Workspace. The importance of this type of event is normally a function of the conceptual depths of a slippage's concepts, and of the size of the Workspace objects involved. As a special case, however, if a slippage is made under the influence of thematic pressure and is compatible with the set of clamped themes, it is deemed to be of very high importance, regardless of the concepts or objects involved.

- *Rule events* record the creation of important rules in the Workspace. The importance of this type of event is a function of the *relative* quality of a rule with respect to all other rules that already exist, with comparatively higher-quality rules being more important.

- *Answer events* record the discovery of new answers, and are always included in the Temporal Trace whenever they occur (*i.e.*, this type of event is deemed to be of maximal importance).

- *Snag events* record the occurrence of snags in the Workspace, and are always included in the Temporal Trace.

- *Pattern-clamp events* record the clamping of theme-patterns, concept-patterns, or codelet-patterns in response to various situations, and are always included in the Temporal Trace.

4.5 Self-Watching

This section explains in detail how Metacat uses the information in the Temporal Trace to monitor its own behavior, and how it can alter its behavior by clamping (or unclamping) various types of patterns in response to this information. Two types of codelets in Metacat are responsible for examining and responding to the events unfolding in the Temporal Trace.

4.5.1 Progress-watcher codelets

The first type of codelet, called a *Progress-watcher*, has two principal functions. First, it is responsible for deciding whether or not to unclamp a set of clamped patterns. If a *Progress-watcher* codelet runs during a clamp period, it examines the *most recent event* in the Trace (which may or may not be the most recent *clamp* event) in order to determine how much time has elapsed since the event occurred. Generally speaking, the purpose of clamping a set of patterns is to precipitate a series of events that reorganize the perceptual configuration of the Workspace in some way (*i.e.*, by causing the creation of new structures). It is therefore better to wait until the structure-building activity occurring in the wake of a clamp has settled down a bit before concluding that the clamp has "run its course". Accordingly, if the amount of elapsed time since the most recent event in the Trace is less than some minimal "settling period", then the codelet simply fizzles, leaving the clamped patterns still in effect. On the other hand, if enough time has passed without any new important events having transpired, the codelet unclamps the patterns and then determines the amount of progress that was made since the clamp occurred. Depending on the amount of progress achieved, the codelet may decide to post a follow-up *Answer-finder* codelet (or an *Answer-justifier* codelet if the program is running in justify mode) in order to see whether a new answer can be made based on the newly-created structures.

The criteria for judging the "success" of a clamp depend on the nature of the clamp itself. Sometimes, the purpose of clamping a set of patterns is to promote the creation of *specific* types of Workspace structures (rules, for example). Most of the time, however, the purpose is to encourage the creation of structures of *any* type, so long as they are compatible with the clamped patterns. For example, the pattern-clamping that occurs during the answer-justification process described in the preceding section is intended to force the creation of mutually-consistent horizontal and vertical mappings, which may depend on building new bridges and groups, making new slippages, and creating new rules. The progress achieved by such a clamp can thus be measured by observing the strengths of the most important Workspace structures that get built in the aftermath of the clamp (since in general the strength of a structure reflects its compatibility with the set of clamped patterns). This information is recorded in the Temporal Trace, in the form of group events, slippage events, and rule events. The progress achieved by the clamp shown in Figure 4.13, for example, can be determined by examining the subsequent group and slippage events. Since these events all represent the creation of structures compatible with the clamped patterns, the degree of progress achieved by this particular clamp is quite high.

In general, associated with every clamp event is a *progress-evaluator* function, which *Progress-watcher* codelets use to evaluate the events in the Trace following the clamp event in order to determine the overall progress achieved by the clamp. Since the progress is evaluated at the end of the ensuing clamp period, only events that occur during this period are considered. In the case of an answer-justification clamp, the evaluator function pays attention to subsequent group events, slippage events, and rule events (in particular to the strengths of the new structures created), but other types of clamps may involve different measures of progress, through the use of different evaluator functions.

In fact, the second principal task performed by *Progress-watcher* codelets involves clamping patterns that focus on the creation of a single type of Workspace structure, rather than on a number of different types, as in the answer-justification process. 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 taking place in the Workspace.

The *Workspace activity*, like temperature, is a simple numerical measure ranging from 0 to 100. However, rather than reflecting structure quality, the activity level provides a quick estimate of the "freshness" of the current Workspace structure configuration. 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 on which to pin the blame, 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 are needed. In the current version of Metacat, these codelets only check to see if new *rules* are needed, but in principle, they could also assess the need for other types of structures, so long as a way of estimating this need could be defined. In the case of rules, the codelets examine the quality of all of the top rules and bottom rules that have been built so far.[9] If either the best top rule or the best bottom rule is still of poor quality, the codelet may try to encourage the creation of better rules by clamping a special codelet-pattern that heightens the urgencies of *Rule-scout*, *Rule-evaluator*, and *Rule-builder* codelets, while simultaneously lowering the urgencies of all other codelet types. Since this type of clamp is only "interested" in the creation of new rules, its progress-evaluator function pays attention only to subsequent rule events in the Trace. The amount of progress achieved is judged solely according to the quality of the rules that get created in the wake of the clamp. Eventually, other *Progress-watcher* codelets will turn off the clamp once enough time has passed without any more events (of any type) being added to the Trace.

4.5.2 Jootser codelets

The second type of codelet in Metacat that "watches the action" from the high-level vantage point of the Temporal Trace is called a *Jootser*. These codelets are responsible

[9] If the program is not running in justify mode, then only the top rules are considered.

for noticing—and breaking out of—repetitive patterns of behavior that the program has fallen into. (See [Hofstadter, 1985b] for a discussion of the notion of *jootsing*, or "jumping out of the system", especially as it relates to the idea of a self-watching computer program.) One example of such behavior, discussed earlier in Chapter 2, involves the program hitting the same snag over and over again. As was sketched in section 2.4.4, a series of identical (or very similar) snag events in the Trace may cause patterns to be clamped in response, which may lead to a new way of looking at things that avoids the snag. However, in addition to snags, *Jootser* codelets are sensitive to other types of repetitive behavior as well. In particular, it is possible for Metacat to become "fixated" on some set of ideas, such that it ends up clamping the same set of patterns over and over again, without making any significant progress. In this case, too, *Jootser* codelets may notice the series of recurring events in the Trace and take action.

Jootsing from repeated snags

As was mentioned earlier, every time an event is recorded in the Temporal Trace, a set of theme-patterns characterizing the event gets recorded along with it. This thematic information can be used as the basis for judging similarity between different events of the same type, such as snags. In the case of snags, however, similarity also depends on whether the snag events involve the same set of Workspace structures (for example, the letter ***z*** and the rule *Change letter-category of rightmost letter to successor* in the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*).

In order to detect repetitive behavior arising from snags, *Jootser* codelets continually scan the Temporal Trace looking for snag events that share the same set of snag structures and have thematic characterizations that overlap to a significant degree. The presence of several such events indicates that the program has run into the same problematic situation several times. To be more precise, the same rule has been translated and then applied unsuccessfully to the target string each time. Since the rule-translation process depends on the way in which the initial string and the target string are perceived as being similar, a new way of looking at these strings (*i.e.*, a new vertical mapping) may be needed in order for Metacat to break out of its rut. Accordingly, if enough similar snag events are detected, a *Jootser* codelet may try to force a reorganization of the vertical mapping by clamping an appropriate vertical theme-pattern. This decision is made probabilistically, depending on the number of snag events and the degree of overlap of their accompanying themes. Unfortunately, *how* to reorganize the vertical mapping so that further snags can be avoided is not clear, unlike in the case of answer-justification, where comparing a pair of rules can provide clues as to which ideas might be useful to try. The best that can be done is to focus on ideas *other* than those that currently characterize the initial–target string similarity.

Consequently, once a *Jootser* codelet has decided to respond to a recurring snag, it creates a vertical theme-pattern consisting of *negatively-activated* themes, based on the themes associated with the snag events, which it then clamps in the Theme-space. In general, however, not all of these associated themes are necessarily "bad themes". For example, the snag in the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"* arises mainly from viewing the ***z*** in ***xyz*** as corresponding to the ***c*** in ***abc***. This idea is characterized by the vertical themes *String-Position: identity* (based on the concept-mapping *rightmost ⇒ rightmost*) and *Object-Type: identity* (based on the concept-mapping *letter ⇒ letter*). Both of these themes are associated with the ensuing snag events, but the source of the problem lies in seeing the components of ***abc*** and ***xyz*** (in particular, ***c*** and ***z***) as occupying identical *positions* in their strings, not in seeing them as being identical types of *objects*. Negatively clamping the *String-Position: identity* theme is thus more likely to remedy the situation than negatively clamping the *Object-Type: identity* theme. In fact, negatively clamping the latter theme may actually interfere with the creation of a new mapping, since any new vertical bridges are likely to be between objects of the same type (*i.e.*, letters), and will thus be severely weakened by the negative *Object-Type: identity* theme. In general, therefore, *Jootser* codelets decide probabilistically which themes to include in the negative theme-pattern that gets clamped in response to a recurring snag.[10]

Once clamped, this theme-pattern exerts negative thematic pressure on codelet processing, encouraging the creation of new Workspace structures *incompatible* with the themes involved in the snag. In order to expedite this process, a special "bottom-up" codelet-pattern is also clamped along with the theme-pattern. This codelet-pattern specifies very high urgencies for all types of bottom-up scout codelets, as well as for their associated evaluator and builder codelets. The effect of this pattern is to accelerate bottom-up exploratory processes, in the hope that a viable alternative to the initial snag-prone interpretation of the problem can be discovered.

Jootsing from repeated clamps

Repeatedly hitting a snag is not the only type of "loopy" behavior to which Metacat is susceptible. Under certain circumstances, it is also possible for the program to fall into a repetitive cycle of *pattern-clamping*. For example, this can occur during answer-justification if clamping a set of patterns in response to unifying a pair of rules fails to produce a vertical mapping supporting the rules. The same set of patterns may end up getting clamped over and over again in a futile attempt to find a coherent

[10]For each theme, this decision is a function of the number of snag events involving the theme (since the more often a theme is associated with hitting a snag, the more likely it is to reflect the source of the problem), and of the conceptual depths of the descriptions related to the theme that are attached to objects directly involved in the snag (since themes that reflect deeper aspects of the snag objects are more likely to be important in characterizing the snag). For example, the snag object in the *"**abc** ⇒ **abd**; **xyz** ⇒ ?"* problem (*i.e.*, the letter ***z***) may be described both in terms of its string position (*i.e.*, *rightmost*) and its object type (*i.e.*, *letter*). Since the conceptual depth of *rightmost* is greater than that of *letter*, the *String-Position* theme has a greater chance of being negatively clamped than does the *Object-Type* theme.

interpretation of the problem based on these rules. Likewise, if an analogy problem happens to involve a string that changes in some difficult-to-describe way, the program may end up repeatedly clamping codelet-patterns in a futile effort to spur the creation of new (or better) rules for describing the change. Repetitive clamping behavior can even arise from unsuccessful attempts to break out of a cycle of snag events. That is, negatively clamping theme-patterns in response to a recurring snag may prove to be ineffective, leading only to further snags and more theme clamping, rather than to a new interpretation of the problem.

Thus, in addition to watching for snag events, *Jootser* codelets also look for recurring clamp events in the Temporal Trace. If enough similar clamp events are noticed, the codelet may decide to respond in a way that depends on the type of clamp involved. Essentially three types of clamps are possible in Metacat:

- *Justify clamps* arise from clamping theme-patterns in response to unifying pairs of rules in justify mode.
- *Rule-codelet clamps* arise from clamping codelet-patterns in an effort to stimulate the creation of new rules.
- *Snag-response clamps* arise from clamping negative theme-patterns in response to a series of recurring snag events.

In the current version of the program, the similarity of clamp events is judged only according to whether or not clamps are of the same type, and whether, in the case of justify clamps, they involve the same pair of rules. The similarity of the patterns involved is not taken into account, although it probably should be, at least in the case of snag-response clamps, since responding to the same snag in different ways by clamping different negative theme-patterns should not be seen as doing the same thing over and over again.[11]

[11] Taking the clamped patterns into account is not really necessary for rule-codelet clamps, since

In any case, faced with several similar clamp events, a *Jootser* codelet decides probabilistically whether or not to "joots" based on the number of clamp events and the average amount of progress that has been achieved by the clamps. (As will be recalled, the progress resulting from a clamp is determined at the end of the ensuing clamp period by *Progress-watcher* codelets, using the progress-evaluator function associated with the clamp.) Generally speaking, the more clamp events there are, the more likely jootsing is to occur, especially if the average amount of progress resulting from the clamps is low. However, in judging the average progress made by a series of similar clamps, the amount of time since each clamp occurred is also taken into account. Thus, if recent clamps seem to be making more progress than earlier clamps, then jootsing is less likely to happen. Conversely, if the clamps appear to be making less and less progress as time goes on, indicating that the ideas represented by the clamped patterns may have exhausted their potential, then the likelihood of jootsing increases.

Unlike jootsing from snags, jootsing from a series of recurring clamp events does not involve the clamping of any new patterns in response. Instead, in the case of recurring rule-codelet or snag-response clamps, Metacat simply "gives up" in a graceful manner and stops.[12] In the case of justify clamps, however, the program's unsuccessful attempts to find a coherent justification for an answer may be due to an inability to make the necessary slippages, even in principle, that are required in order to justify the answer.

For example, if the program is given the problem "***xqc*** ⇒ ***xqd***; ***mrrjjj*** ⇒ ***?***" and asked to justify the answer ***mrrjjjj***, it may end up repeatedly clamping patterns in a futile attempt to make a *Letter-Category* ⇒ *Length* slippage, which is required in

these clamps always involve exactly the same codelet-pattern.

[12] Upon quitting, the program politely excuses itself with the message "Excuse me—I think I'll go get some more punch", as any tactful person might do in order to escape from an interminable bore at a party. See [Hofstadter, 1985b].

order to relate the rule describing ***xqc*** ⇒ ***xqd*** (*i.e.*, *Change letter-category of rightmost letter to successor*) to the rule describing ***mrrjjj*** ⇒ ***mrrjjjj*** (*i.e.*, *Increase length of rightmost group by one*[13]). Unfortunately, no such slippage is possible because ***xqc*** cannot be seen as a single group based on letter-categories, and thus no bridge can be made between ***xqc*** and ***mrrjjj*** as a whole. The other required slippage, *letter* ⇒ *group*, presents no problem, however, since a bridge can easily be made between the letter ***c*** of ***xqc*** and the ***jjj*** group of ***mrrjjj***. Thus, in translating the top rule, the closest Metacat can come to matching the bottom rule is *Change letter-category of rightmost group to successor*—just one slippage away from a successful justification of ***mrrjjjj***.

In general, once the program has recognized that it has fallen into a repetitive cycle of justify clamping, it may decide to settle for an *unjustified* interpretation of an answer, depending on how close it can come to justifying it legitimately. Thus, if rules supported by the appropriate horizontal mappings 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 repeated failure to understand how the unjustified slippages arise. The more unjustified slippages that remain, however, the less likely jootsing is to occur. Furthermore, there is always the possibility that the program will give up on an answer too easily, reporting it as unjustified when in fact it could be fully justified with further effort, but in practice this does not happen very often. On the other hand, of course, it is impossible for Metacat to know which answers in principle are beyond its ability to justify (such as the answer ***mrrjjjj*** in the earlier problem), 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

[13] Another way of expressing this rule in English is *Change length of rightmost group to successor*, which brings out the differences between the rules a little more clearly.

seeing ***xqc*** as a group based on letter-categories). In any case, the program is at least aware of the fact that it has settled for an unjustified answer, and notes this fact, along with the slippages that it failed to justify, in its memory.

One last point is worth mentioning. Metacat's ability to eventually give up in response to recurring justify clamps or to recurring rule-codelet clamps in a sense represents *first-order* jootsing, because in both cases the recurring clamp events arise from circumstances in the Workspace—in the former case, from pairs of rules that can be unified, and in the latter case, from a lack of sufficiently high-quality rules. In other words, these types of clamp events arise from patterns of activity at the subcognitive processing level. Likewise, snag events also arise from subcognitive processing activity. Accordingly, Metacat's ability to respond to recurring snag events (by clamping a negative theme-pattern, rather than just giving up) also represents first-order jootsing—or at least attempted first-order jootsing, since clamping the pattern may not in fact help to avoid further snags.

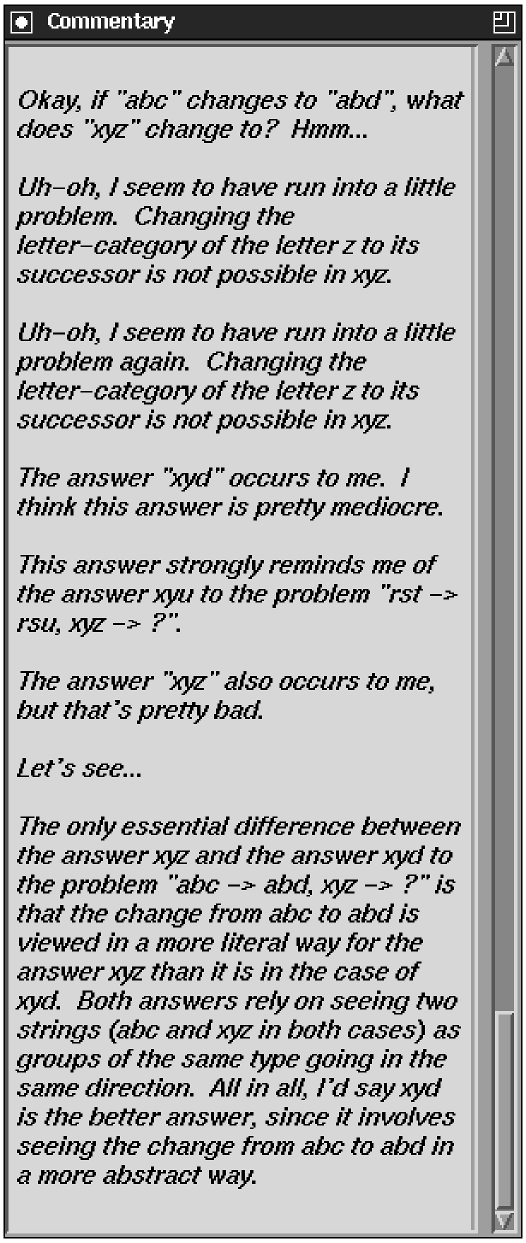
In contrast, the program's ability to respond to a recurring sequence of ineffective *snag-response clamps* (by giving up) represents *higher-order* or *meta-level* jootsing (*i.e.*, jootsing from repeated unsuccessful jootsing), because the recurring snag-response clamps arise from activity in the Temporal Trace—that is, from sequences of recurring snag events. In other words, snag-response clamps 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*. The important point is that the same general mechanisms (*i.e.*, *Jootser* codelets and the explicit representation of processing events in the Trace) are responsible for both first-order and meta-level jootsing in Metacat. This reflects the belief that no fundamental distinction should be made between the different levels of a self-watching system. That is, all levels of such a system should be fused, rather than being organized into a rigid hierarchy, with each level distinct from the rest and responsible only for watching and responding to activity occurring at the level immediately below. (See [Hofstadter, 1985b] for a full discussion of these ideas.)

4.6 The Comment Window

As Metacat works on an analogy problem, watching its own behavior in the process, it displays a running commentary in English of its ideas and observations about the problem and about its own "train of thought". This narrative, which appears in Metacat's *Comment Window*, is not an event-by-event transcription of the information appearing in the Temporal Trace, although it does, of course, correspond closely to the chain of events recorded there. Rather, it simply consists of messages generated by codelets under a variety of different circumstances as they go about their business. Essentially, this amounts to the program "thinking out loud" while it works on a problem. When Metacat encounters a snag, for instance, it reports this fact in the Comment Window 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 that happen to "come to mind" as a result.[14] The program also mentions when it is getting "frustrated" by a lack of progress. Furthermore, if it hits on some new idea to try, it gives a brief assessment of the progress achieved, in retrospect, as a result of focusing on the idea. As will be explained in section 4.7.3, the program can also comment on the similarities and differences between various answers, if asked to do so by the user.

Figure 4.14 illustrates the type of commentary typically generated by the program during a run. The first example shows a run of the problem ***"abc*** $\Rightarrow$ ***abd; xyz*** $\Rightarrow$ ***?"*** in which the program hits the usual ***z*** snag a couple of times and then answers ***xyd***. (In fact, this same run was shown earlier in Temporal Trace form in Figure 4.12.) As

[14]Metacat's ability to be reminded of other answers will be discussed in section 4.7.5.



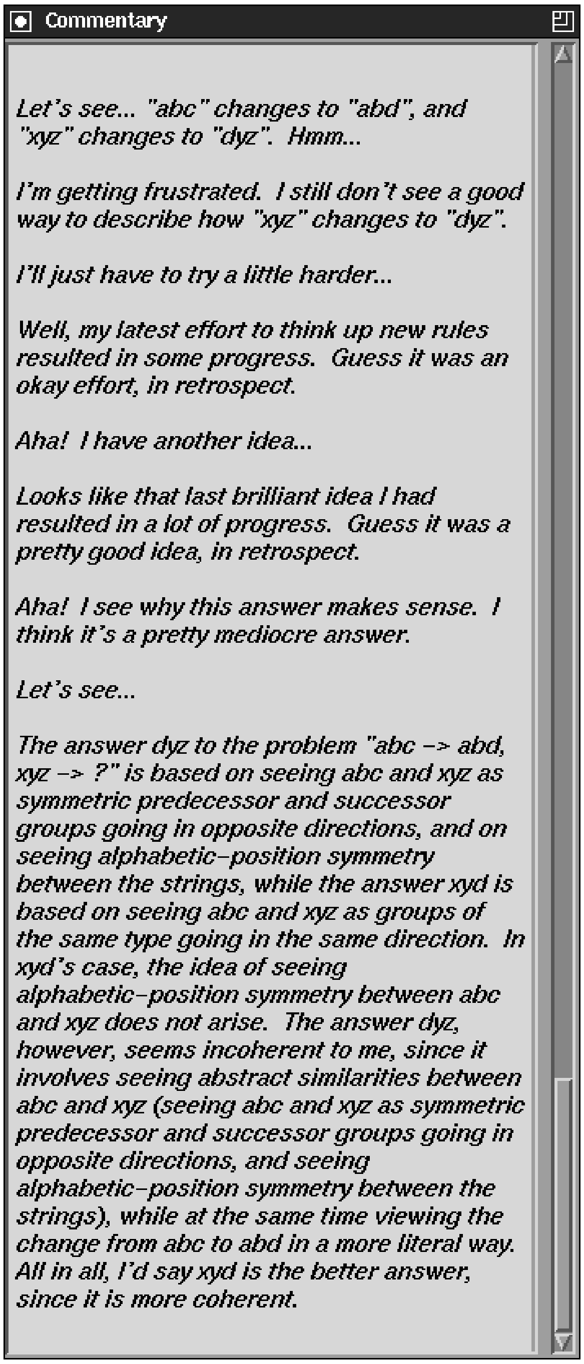


Figure 4.14: *Metacat's running commentary for a run of the problem "**abc** ⇒ **abd**; **xyz** ⇒ **?**" in which it found the answers **xyd** and **xyz** (left), and for a justification run of the same problem in which the program was given the answer **dyz** (right).*

it happens, the answer ***xyd*** reminds the program of a similar answer to a different problem that it has already encountered. Continuing on, the program then finds the "do-nothing" answer ***xyz*** (based on the rule *Change letter-category of letter 'c' to 'd'*). At this point, prompted by the user[15], the program compares the answer ***xyz*** to the answer ***xyd***, expressing a preference for the latter answer.

The second example shows Metacat justifying the answer ***dyz*** for the same problem. In this run, the program encounters some difficulty at first in building a rule for describing how ***xyz*** changes to ***dyz***. Its comment about "trying harder" arises from clamping a codelet-pattern in order to stimulate the creation of new rules. As it turns out, three new rules get created in the wake of this clamp. The program therefore regards the amount of progress made by the clamp as satisfactory. In fact, this enables the program to subsequently unify a pair of rules, which leads to a second round of "brainstorming" (*i.e.*, a justify clamp). This clamp spurs the creation of many new structures, leading to the re-interpretation of ***abc*** and ***xyz*** as "mirror images" of each other, which in turn leads to a successful justification of ***dyz***. The program thus considers the progress achieved by the clamp to be very high, even though it considers ***dyz*** itself to be a pretty mediocre answer. Finally, the program is asked to compare this answer to the answer ***xyd***, 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. In fact, the program possesses *no genuine linguistic ability whatsoever*; its ability to "speak" is purely an illusion arising from a flexible set of phrase-templates, rather than from a flexible command of English. These phrase-templates get filled in and combined in complicated but purely mechanical ways, according to the circumstances at hand. For example, in the first run shown in Figure 4.14, the explanation of the snag is

[15]The program prints the phrase "Let's see . . . " whenever it is prompted to compare two answers.

generated on the basis of the concepts and Workspace structures involved in the snag—namely, the Slipnet concept *Letter-Category*, the letter ***z***, the Slipnet concept *successor*, and the Workspace string ***xyz***. 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 prefabricated phrases to describe various numerical measures—such as the progress achieved by a clamp—which are chosen from lists of possible alternatives on the basis of the numerical values involved. For instance, in the second run shown, the phrases "some", "an okay", "a lot of", and "a pretty good" are all chosen on the basis of the underlying numerical progress values associated with the clamps that occur during the run. Other sentences are completely canned, such as "I'm getting frustrated" and "I'll just have to try a little harder…", which the program prints out whenever it clamps a codelet-pattern in search of better rules, or "Aha! I have another idea…", which it prints out whenever a justify clamp occurs. The commentary generated by the program when comparing different answers is produced in a similarly mechanical fashion. (The particular way in which the program generated its commentary about the similarities and differences between ***dyz*** and ***xyd*** shown in Figure 4.14 will be discussed in detail in section 4.7.4.) Furthermore, no type of *linguistic interaction* with the program—in any form—is possible. For instance, "asking" the program to compare two answers is accomplished simply by clicking on graphical icons associated with the answers.

Metacat's English-language veneer, although deceptive in a certain sense, is not intended to deceive. Rather, it is intended simply to show the various things that happen during the course of a run, in a somewhat whimsical but also very user-friendly fashion. In the case of comparing two answers, it is intended to show, in an easily-understandable way, the various parallels and distinctions between the answers that are recognized by the program. As will be discussed in the following sections, answers are compared on the basis of their underlying conceptual representations, which consist mainly of themes and Slipnet concepts. Metacat's ability to compare answers *at this representational level* is what counts, not its ability to generate English-language summaries of these comparisons.

That said, it is worth adding that not *all* of the words used by the program verge on being completely devoid of semantic content. To be sure, most of them do (*e.g.*, "okay", "frustrated", "try", "mediocre", "I", "me", and so on). However, a few of them, such as "successor", "opposite", "direction", "alphabetic-position", "groups", and "letter", reflect concepts that the program *does* understand—in a more genuine, and quite defensible, sense—about its letter-string microworld.

In any case, as was indicated above, the chatty, colloquial tone of Metacat's commentary is meant to be humorous more than anything else. However, it is also important at this point to acknowledge the potential dangers 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. (For a discussion of Joseph Weizenbaum's ELIZA program, from which the Eliza effect takes its name, see [Weizenbaum, 1976].) Clearly, the output generated by Metacat might easily lead (or mislead) a casual observer into falling for this effect. Therefore, in the interest of transparency, the current version of the program can be run in two different linguistic output modes.

When running in "Eliza mode", Metacat generates the type of informal commentary shown earlier. With this mode turned off, however, the program uses more neutral language to describe the events that occur during a run. Figure 4.15 shows the earlier output from the two runs of Figure 4.14, together with the more neutral output generated during the same two runs with Eliza mode turned off.[16] As can be

[16]Turning off Eliza mode, however, does not affect the commentary generated by the program when comparing different answers. Thus the program's descriptions of the similarities and differences between ***xyd***, ***xyz***, and ***dyz*** are the same in each case.

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.

Beginning run: If "abc" changes to "abd", what does "xyz" change to?

Hit a snag: Changing the letter-category of the letter z to its successor is not possible in xyz.

Hit another snag: Changing the letter-category of the letter z to its successor is not possible in xyz.

Found the answer "xyd". Answer quality = 73.

This answer is reminiscent of the answer xyu to the problem "rst -> rsu, xyz -> ?". Reminding strength = 80.

Found the answer "xyz". Answer quality = 57.

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 4.15: *Metacat's commentary for the same two runs shown in Figure 4.14, but with "Eliza mode" turned off for comparison. Comments on the right correspond to comments on the left on a one-to-one basis. The program's explanations of the differences between answers are the same as before, and thus are not shown.*

seen from the figure, Metacat generates exactly the same number of paragraphs in either mode, which emphasizes the fact that the commentary produced by the program when running in one mode is isomorphic to the commentary produced in the other mode. In fact, a side-by-side comparison of the output generated in different modes on a particular run is quite revealing in a way, because it brings out more clearly which aspects of the program's commentary are mere "window-dressing", and which aspects actually convey meaningful information about the run.

4.7 The Episodic Memory

The preceding examples convey the flavor of Metacat's ability to "talk" about its answers, and about its own behavior, in various ways. Clearly, much is going on beneath the surface here. In particular, the program's capacity to recall previously-encountered answers, and to explain the similarities and differences that exist between answers, relies on storing abstract representations of answers in long-term memory. The remainder of this chapter describes these representations and how they allow Metacat to compare and contrast its answers in an insightful manner.

4.7.1 Answer descriptions

When Metacat discovers a new answer, a considerable amount of information typically exists—in the form of various types of structures—in the Workspace, the Themespace, and the Temporal Trace. Usually, though, not all of this information is relevant to the newly-created answer. For instance, extraneous groups, bridges, or rules, which play no role in the creation of the answer, may exist in the Workspace. Likewise, a number of partially-activated—or even dominant—themes characterizing ideas of merely peripheral importance to the answer may exist in the Themespace. Furthermore, many of the processing events appearing in the Trace may have nothing to do with the answer, or may refer to structures that no longer exist.

Therefore, in order to create a representation of the answer suitable for storing in long-term memory, Metacat must "distill" from this welter of information an abstract *answer description* that captures the essence of the answer without including all of the information that is available. This description must summarize the key factors that led to the answer's creation—namely, the ways in which the various strings involved in the problem were perceived as being similar (and different). These factors can be represented by a set of theme-patterns and rules. Specifically, an answer description is composed of the following structures:

- The *Workspace structures* directly involved in creating the answer. This includes groups, bridges, and concept-mappings, as well as the letter-strings themselves.

- A *vertical theme-pattern* representing the similarity perceived between the top situation and the bottom situation (*i.e.*, between the initial string and the target string).

- A *top theme-pattern* representing the similarity perceived between the initial string and the modified string.

- A *top rule* representing the way in which the initial string is perceived as changing to the modified string.

- A *bottom theme-pattern* representing the similarity perceived between the target string and the answer string (when running in justify mode).

- A *bottom rule* representing the way in which the target string is perceived as changing to the answer string (when running in justify mode).

- An *unjustified theme-pattern* representing any unjustified slippages that the program failed to come to terms with in trying to make sense of an answer provided to it (when running in justify mode).

The top and bottom theme-patterns of an answer description are just the theme-patterns associated with the top and bottom rules, which were created at the time the rules were built. To create the vertical theme-pattern, Metacat examines the activations of vertical themes in the Themespace, along with recent group and slippage events appearing in the Temporal Trace. If slippages have recently been made—especially slippages involving whole-string groups—the themes associated with these slippage events will be included in the answer description, since the ideas they represent are likely to have played a central role in creating the answer.[17] (Slippages that do not appear in the Trace may also have occurred, but they can be safely ignored, since they were not considered to be of sufficient importance in the first place to deserve explicit representation in the Trace.) Furthermore, identity concept-mappings involving whole-string groups are also likely to be important, since they characterize similarities between highly-chunked situations, so themes associated with any such concept-mappings are included as well.[18]

As an example, Figure 4.16 shows the full answer description created for the answer ***wyz*** to the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*. This particular example arises from the answer-justification run described earlier in section 2.4.5 of Chapter 2. The final Workspace configuration of the run appears in Figure 2.5, and the sequence of events recorded in the Temporal Trace during the run is shown in Figure 4.13. In particular,

[17] In general, associated with every slippage event in the Trace is a theme consisting of the concepts involved in the slippage. For instance, a vertical *Alphabetic-Position: opposite* theme would be associated with a vertical *first ⇒ last* slippage.

[18] However, a few caveats are in order here. In the current version of the program, only certain types of vertical themes are allowed to appear in answer descriptions. Specifically, only vertical themes of the category *String-Position*, *Alphabetic-Position*, *Direction*, *Group-Type*, or *Bond-Facet* are permitted. Furthermore, in the case of *Bond-Facet*, no identity themes are allowed (*i.e.*, only *Bond-Facet: different* themes are permitted). These constraints make it easier for the program to compare and contrast its answers, but they are essentially artificial and thus unsatisfactory. This point will be discussed further in Chapter 6.

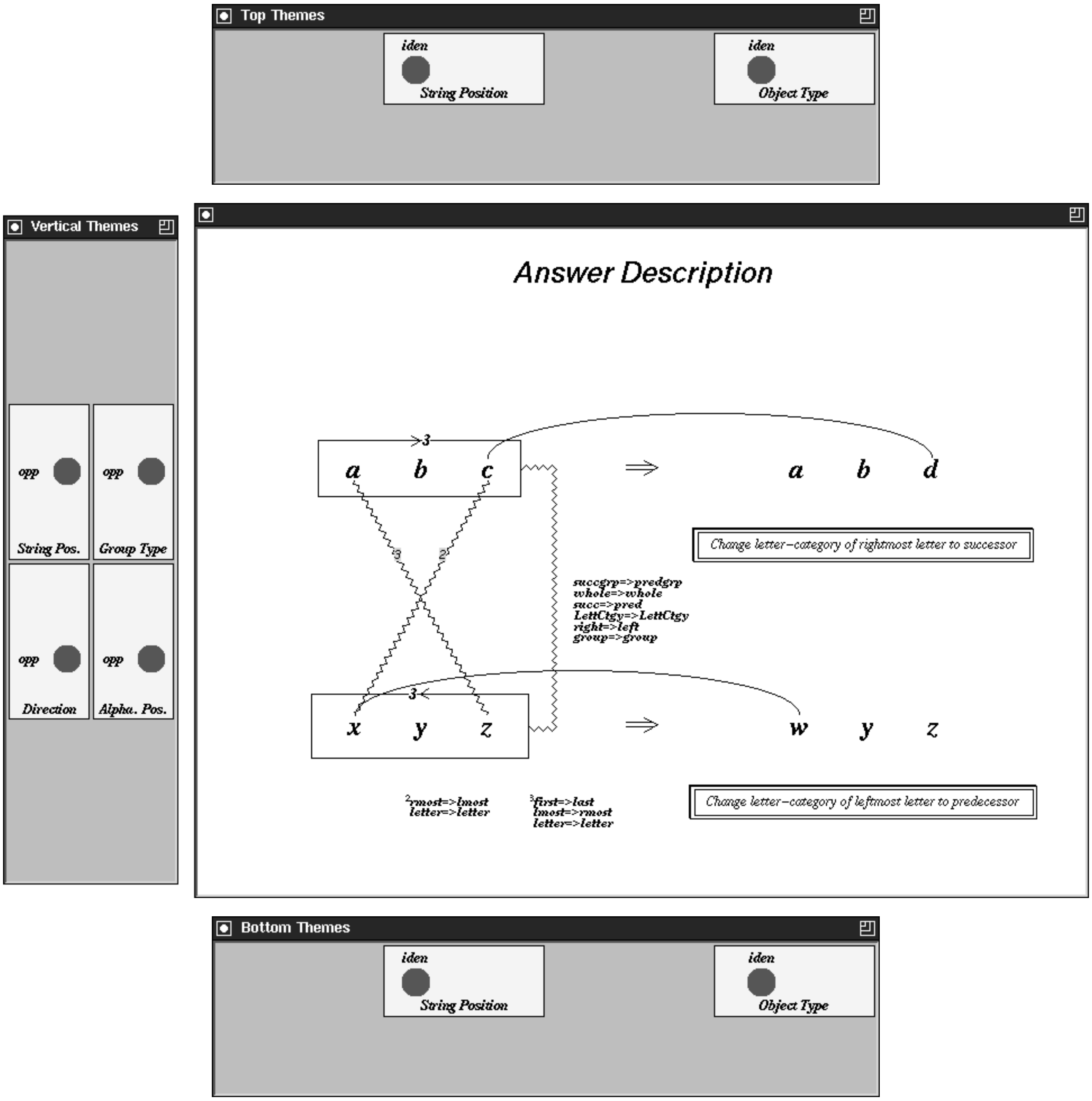


Figure 4.16: *The full answer description created for the answer* ***wyz*** *to the problem "****abc****⇒* ***abd****;* ***xyz****⇒* ***?****", showing the themes, rules, and other structures involved.*

the vertical theme-pattern appearing in the answer description is abstracted from the series of slippage events recorded in the Trace.

Finally, in the case of an unjustified answer, an additional theme-pattern is created from the answer's unjustified slippages representing those aspects of the answer that remain "unsupported by the evidence". For example, the answer description for the unjustified answer ***mrrjjjj*** to the problem *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ **?**"* (discussed earlier in section 4.5.2) would include an unjustified vertical *Bond-Facet: different* theme, reflecting the failure of the program to make the required *Letter-Category ⇒ Length* slippage between the top string ***xqc*** and the bottom string ***mrrjjj***. This theme-pattern, like the other patterns of an answer description, serves as a basis for judging the relative quality of an answer as compared with other answers the program has encountered.

4.7.2 Snag descriptions

In addition to remembering its answers, Metacat also remembers the snags that it encounters while solving problems on its own. On hitting a new snag for the first time, the program creates an abstract *snag description* that characterizes the situation (in addition to creating a new snag event in the Temporal Trace), which it then stores in memory. Like answer descriptions, snag descriptions consist of themes and Workspace structures (*i.e.*, those directly responsible for causing the snag), and are used by the program in comparing and contrasting answers with one another. Specifically, a snag description consists of the following structures:

- The *Workspace structures* directly involved in the snag, including groups, bridges, and concept-mappings, as well as the letter-strings themselves.
- A *vertical theme-pattern* characterizing the way in which the objects directly responsible for the snag in the bottom situation are perceived as being similar to

their counterpart objects in the top situation. This theme-pattern is based on the concept-mappings underlying the bridges associated with the snag objects.

- A *top rule* representing the way in which the initial string is perceived as changing to the modified string.

- The *translated rule* leading to the snag itself.

Unlike answer descriptions, snag descriptions do not include any top or bottom theme-patterns.

Storing snag descriptions in long-term memory gives Metacat the ability to "appreciate" certain answers in ways that otherwise would not be possible. For example, consider the problem *"**eqe** ⇒ **qeq**; **abbbc** ⇒ **?**"* [Hofstadter and FARG, 1995, pp. 305–306]. In this problem, ***eqe*** gets, in a sense, turned inside-out, an idea that can be captured—at least approximately—by the rule *Swap letter-categories of all objects in string*. However, it is not so easy to do "the same thing" to ***abbbc***, since three different letter-categories are involved, instead of just two. One particularly elegant way out of this quandary is to reperceive ***abbbc*** as *1–3–1* and then swap the group-lengths rather than the letter-categories, yielding the answer ***aaabccc***. Unfortunately, Metacat is unable to get this answer on its own, because it cannot see ***eqe*** and ***abbbc*** as single chunks based, respectively, on the ideas of letter-category and group-length, and is thus unable to make the required *Letter-Category ⇒ Length* slippage. Instead, it ends up repeatedly hitting a snag in trying to "swap" the letter-categories of ***a***, ***bbb***, and ***c***. On the other hand, if given this answer at the outset, it can *almost* make sense of it—save for the unjustified *Letter-Category ⇒ Length* slippage.

The same goes for the answer ***aaabaaa*** to the problem *"**eqe** ⇒ **qeq**; **abbba** ⇒ **?**"*. Metacat can (almost) justify this answer, but cannot get it on its own. However, there is a crucial difference between ***aaabaaa*** and ***aaabccc***. In the problem *"**eqe** ⇒ **qeq**;*

abbba $\Rightarrow$ ***?***", no good reason exists to see ***abbba*** as *1–3–1*, since swapping letter-categories is perfectly feasible. That is, no snag arises in this problem. In a sense, then, the answer ***aaabccc*** is "justified" after all (since seeing ***abbbc*** as *1–3–1* avoids a snag), while ***aaabaaa*** is not (since seeing ***abbba*** as *1–3–1* is unnecessary). As will be seen in the next chapter, Metacat can make this observation, but it can only do so if it knows that the problem *"**eqe** $\Rightarrow$ **qeq**; **abbbc** $\Rightarrow$ **?**"* normally leads to a snag. If it has tried this problem on its own, it will know this, because the appropriate snag description will exist in memory. (Conversely, if it is asked to justify the answer ***aaabccc*** without having first attempted the problem itself, it will remain unaware of the possibility of a snag arising, since snags never arise during answer-justification.) In this way, snag descriptions enrich the program's capacity for understanding its answers.

4.7.3 Comparing and contrasting answers

When Metacat is asked to compare two answers it has encountered, it retrieves the abstract descriptions of the answers from its Episodic Memory and analyzes the themes and rules contained in these descriptions. In general, two answers may have identical themes in common (called *common themes*), they may have themes which share the same category but differ by relation (called *differing themes*), or one or both answers may have themes that are not shared by the other answer at all (called *unique themes*). Furthermore, some of these themes may be unjustified for one of the answers but not for the other. Of course, various combinations of these types of themes are also possible for an answer.

For example, referring back to the answer descriptions shown in Table 2.1 on page 67, it can be seen that the answers ***xyd*** and ***xyu*** share a common *String-Position: identity* vertical theme. On the other hand, ***xyu*** and ***uyz*** are based on the differing themes of *String-Position: identity* and *String-Position: opposite*. Finally, in the case of the two ***wyz*** answers, the first answer involves a unique *Alphabetic-Position: opposite* theme.

Another possibility, discussed at the end of the preceding section, is that an unjustified theme underpinning an answer may in fact turn out to be "justified" in a certain sense, if the theme represents an idea that enables a snag to be avoided. Some of an answer description's unjustified themes may therefore be reclassified as *snag-justified themes.* For example, when Metacat gives up on justifying the answer ***aaabccc*** to the problem *"**eqe** ⇒ **qeq; abbbc** ⇒ **?**"*, it includes an unjustified *Bond-Facet: different* theme[19] in its description of ***aaabccc***, on account of its failure to make the necessary *Letter-Category ⇒ Length* slippage. Likewise, it includes the same unjustified theme in its description of the answer ***aaabaaa*** to the problem *"**eqe** ⇒ **qeq; abbba** ⇒ **?**"*, for the same reasons. However, upon comparing these two answers, it considers the *Bond-Facet: different* theme to be justified by the possibility of a snag in the case of the former answer—provided that it has encountered this snag on its own before—but to be unjustified in the case of the latter answer.

To be more precise, the presence of a snag description in memory involving exactly the same letter-strings and rule as some answer description indicates that the program has tried this problem on its own before—using exactly the same rule—and run into a snag. Thus the *differences between the themes* involved in the snag and the themes involved in the answer provide a strong clue as to how, in the case of the answer, the snag is avoided, since everything else about the two descriptions is the same. To bring this out more clearly, Table 4.1 shows the answer descriptions for ***aaabaaa*** and ***aaabccc***, as well as the snag description that arises from trying to swap the letter-categories of ***abbbc*** in the problem *"**eqe** ⇒ **qeq; abbbc** ⇒ **?**"*.[20] In

[19]A *Bond-Facet: different* theme represents the idea of viewing the components of two strings (or groups) as being bonded together differently—on the basis of letter-categories in one case and group-lengths in the other.

[20]A few themes unnecessary for the purposes of the example have been omitted from this table for the sake of clarity.

Problem/Answer	Vertical Theme	Unjustified Theme
eqe $\Rightarrow$ ***qeq; abbba*** $\Rightarrow$ ***aaabaaa***	*String-Position: identity*	*Bond-Facet: different*
eqe $\Rightarrow$ ***qeq; abbbc*** $\Rightarrow$ ***aaabccc***	*String-Position: identity*	*Bond-Facet: different*
eqe $\Rightarrow$ ***qeq; abbbc*** $\Rightarrow$ ***SNAG***	*String-Position: identity*	
Rule: *Swap letter-categories of all objects in string*		

Table 4.1: *Two answer descriptions and one snag description for the problems "**eqe** $\Rightarrow$ **qeq; abbba** $\Rightarrow$ **?**" and "**eqe** $\Rightarrow$ **qeq; abbbc** $\Rightarrow$ **?**", showing the various themes involved. The same rule is included in all three descriptions.*

each case, the vertical *String-Position: identity* theme arises from aligning the target string with ***eqe*** in a straightforward way. Likewise, the same "swapping" rule is involved in each case. By comparing the answer description for ***aaabccc*** with the snag description, the key to avoiding the snag in this problem becomes clear: it is the idea of seeing ***eqe*** and ***abbbc*** as being "glued together" in different ways (*i.e.*, according to letter-categories in one case and group-lengths in the other), represented by the *Bond-Facet: different* theme. Therefore, instead of considering this theme to be unjustified, Metacat considers it to be "snag-justified". In contrast, this theme remains unjustified for the answer ***aaabaaa***, since no corresponding snag description exists for the problem "**eqe** $\Rightarrow$ **qeq; abbba** $\Rightarrow$ **?**". In this way, Metacat can perceive the critical difference between ***aaabaaa*** and ***aaabccc***, even though the themes associated with each answer are identical.

In addition to comparing the themes associated with answers, the program also compares the accompanying rules, both structurally and in terms of their overall levels of abstractness.[21] Essentially, comparing two rules involves "aligning" them in order to highlight any differences that may exist in their internal structure, or between the various concepts making up the rules. (This is similar to the process of rule unification described in section 4.3.1.) Furthermore, the *coherence* of an answer can

[21]Rule abstractness is discussed in section 3.3.5 of Chapter 3.

Problem/Answer	Vertical Themes	
abc ⇒ ***abd***; ***xyz*** ⇒ ***dyz***	*String-Position: opposite*	*Direction: opposite*
	Group-Type: opposite	*Alphabetic-Position: opposite*
abc ⇒ ***abd***; ***xyz*** ⇒ ***xyd***	*String-Position: identity*	*Direction: identity*
	Group-Type: identity	

Table 4.2: *The vertical themes associated with the answers **dyz** and **xyd** to the problem "**abc** ⇒ **abd**; **xyz** ⇒ **?**".*

be checked by comparing the abstractness of the answer's rule with the abstractness of the themes associated with the answer, as determined by the average abstractness of the themes' constituent concepts. For example, the answer ***dyz*** to the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"* involves themes based on 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 incoherent, as the program itself explained (in a somewhat convoluted manner) in Figure 4.14.

4.7.4 Generating commentary in English: An example

The precise way in which Metacat generated its commentary about the similarities and differences between ***dyz*** and ***xyd*** shown in Figure 4.14 will now be described in detail. The starting point for comparing ***dyz*** to ***xyd*** is the set of vertical themes associated with each answer (see Table 4.2).[22] The answer description for ***dyz*** includes the three differing themes *String-Position: opposite*, *Direction: opposite*, and *Group-Type: opposite*, while that of ***xyd*** includes the differing themes *String-Position: identity*, *Direction: identity*, and *Group-Type: identity*. In addition, a unique *Alphabetic-Position: opposite* theme is also included in ***dyz***'s answer description. These themes,

[22]In the current version of Metacat, neither top themes nor bottom themes are used to compare answers. Ideally, of course, answers should be compared on the basis of all of their associated themes. Nevertheless, as will be seen in the next chapter, many answers can be insightfully contrasted on the basis of their vertical themes alone.

together with the rules associated with each answer, are used by the program in deciding which phrase-templates to include in its commentary, and how to fill them in. The first such template is shown below:

The answer __1__ *is based* __2__ __3__, *while the answer* __4__ *is based* __5__ __6__ .

This particular template is chosen because differences exist between the themes. Both ***dyz*** and ***xyd*** include themes that are not present in the other answer, so the general form "The answer ..., while the answer ..." is used, in order to contrast their differences. In general, the first time an answer is mentioned, its full description is used, so slot 1 gets filled in with "***dyz*** to the problem '***abc*** ⇒ ***abd***, ***xyz*** ⇒ ***?***'". Furthermore, if common themes exist for an answer (in addition to the themes being described), then the phrase "in part" is added in slot 2. However, in ***dyz***'s case, no such themes exist, so this slot is left blank. Slot 3 is more complicated. This slot gets filled in with a phrase describing all of ***dyz***'s differing and unique themes. In general, since an answer may have more than one such theme (as in ***dyz***'s case), any number of subphrases may appear here, separated by commas. These subphrases are in turn constructed from various templates, which are chosen on the basis of the themes being described. In the present case, two such templates are involved—one for describing the themes *String-Position: opposite*, *Direction: opposite*, and *Group-Type: opposite*, and one for describing the theme *Alphabetic-Position: opposite*. The first of these templates is shown below:

__1__ *see*__2__ __3__ *as* __4__ __5__ __6__

In general, slots 1 and 2 take various prepositions and verb endings, chosen according to circumstances, in order to yield grammatical English. In the present example, these slots are used to construct the phrase "on seeing". Next, the phrase "***abc*** and ***xyz***" appears in slot 3, which describes the initial string and target string associated with ***dyz***. The *Group-Type: opposite* theme itself is described using the stock phrase "symmetric predecessor and successor groups", which appears in slot 4. Likewise, the *Direction: opposite* theme is described with the phrase "going in opposite directions" in slot 5, which makes explicitly mentioning the *String-Position: opposite* theme unnecessary. The last slot is reserved for various caveats that may need to be included as well. For example, in the case of unjustified themes, the phrase "(although there is no good reason for doing so)" would be added. In the case of snag-justified themes, the phrase "(which avoids a snag that would otherwise arise from the fact that ...)" would be added, with an explanation of why the snag occurred inserted in place of the ellipsis, such as "changing the letter-category of the letter ***z*** to its successor is not possible in ***xyz***". In ***dyz***'s case, however, no unjustified themes exist, so no caveats are necessary.

The template used to describe ***dyz***'s *Alphabetic-Position: opposite* theme is similar to the above template, and is shown below:

__*1*__ *see* __*2*__ *alphabetic-position* __*3*__ *between* __*4*__ __*5*__

As before, the first two slots are used to create the phrase "on seeing". Since the theme is based on the concept of *opposite*, the word "symmetry" is inserted in slot 3. (The word "sameness" would be used for an identity theme.) Slot 4 describes the initial string and target string, but in this case, the phrase used is simply "the strings", since ***abc*** and ***xyz*** have already been explicitly mentioned in the earlier template. The last slot, as before, is reserved for caveats.

This completes the description of the themes associated with ***dyz***. Let us now shift back to the original top-level template, whose slots 4, 5, and 6 get filled in according to ***xyd***'s themes in a similar fashion. The phrases "groups of the same type" and "going in the same direction", however, are used here, on account of the identity themes involved. Furthermore, the answer ***xyd*** itself is described in an abbreviated form, since the full problem has already been mentioned in the description of the rival answer ***dyz***.

The next sentence appearing in the commentary points out the uniqueness of ***dyz***'s *Alphabetic-Position: opposite* theme. In general, if an answer has unique themes, a sentence of the form shown below is included:

In __1__, the idea __2__ does not arise.

In this example, the phrase "***xyd***'s case" appears in slot 1, followed by the description of the theme in slot 2. The latter phrase is constructed in the same manner as before, except that here "of seeing" is used instead of "on seeing". (If several unique themes exist, descriptions for all of them are constructed and included in slot 2, separated by commas.)

As was explained earlier, Metacat considers the answer ***dyz*** to be incoherent. To express this fact, another sentence is added to the commentary, based on the following template:

The answer __1__, however, seems incoherent to me, since it involves seeing __2__ between __3__ (__4__), while at the same time viewing __5__ in a more literal way.

Phrases for the names of the strings involved go in slots 1 and 3 ("***dyz***" and "***abc*** and ***xyz***", respectively). The phrase appearing in slot 2 is either "an abstract similarity" or "abstract similarities", depending on the number of themes associated with the incoherent answer. In ***dyz***'s case, the plural phrase is used. Slot 4 contains essentially the same phrases used earlier to describe ***dyz***'s themes, although here the word "on" is omitted. This phrase makes clear which types of abstract similarities are being referred to, although this may make the English sound slightly redundant and unnatural. The phrase appearing in the last slot refers to the initial string and the modified string, and is constructed by inserting the names of these strings ("***abc***" and "***abd***") into the template "the change from ... to ...".

Finally, the program states its overall "preference" for ***xyd*** by filling in the following template:

All in all, I'd say _1_ is the better answer, _2_ .

In general, the name of the preferred answer appears in the first slot, followed by a phrase describing the program's reason for preferring this answer. In this case, the phrase is "since it is more coherent". Other phrases, however, are possible under different circumstances. For example, if one answer involves unjustified themes while the other does not, the program will express a preference for the latter answer with the phrase "since it involves no unjustified ideas" in slot 2. On the other hand, if no unjustified themes exist, but one answer involves *more* themes than the other, the program will prefer the latter answer, expressing this preference with the phrase "since it is based on a richer set of ideas".

And so on. In short, many phrase-templates exist, not all of which are described here (as can be seen from the program's commentary on ***xyz*** and ***xyd*** shown in the first run of Figure 4.14). These templates can be combined in many intricate ways to yield quite plausible-sounding English commentary. From a theoretical standpoint, however, this surface-level expressive capacity of the program is fundamentally *uninteresting*. What *is* interesting is Metacat's deeper capacity to recognize subtle similarities and differences between answers on the basis of the common themes, the differing themes, the unique themes, the unjustified themes, the snag-justified themes, and the rules that constitute answer descriptions. This analysis is carried out at an abstract representational level, not at a linguistic level—a distinction that must be kept in mind when judging the output of the program.

4.7.5 Reminding

Closely related to the issue of answer comparison is the phenomenon of reminding, in which one answer may trigger the spontaneous retrieval from long-term memory of other answers that are in some way similar to the current answer. In Metacat, this may happen whenever a new answer is discovered (or justified) by the program.

Every answer description stored in memory has an associated *activation* level ranging from 0 to 100. Whenever a new answer event occurs in the Temporal Trace, each answer description computes the *distance* between itself and the new answer description created from the information in the Trace, updating its level of activation according to the distance. If the activation level exceeds some threshold, Metacat will be reminded of the answer, to the extent that the threshold is exceeded. In other words, the activation level of an answer reflects how strongly the program is reminded of it.[23]

For example, Figure 4.17 shows Metacat's memory upon discovering the answer ***wyz*** to the problem *"**rst**⇒ **rsu**; **xyz**⇒ **?**"*, after having encountered several other answers to this problem and to the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"* (in addition, a snag description for the latter problem also exists). The activation levels of answers are indicated by shades of grey, ranging from white for the most strongly-activated answers to dark grey for dormant answers—so that more weakly-activated answers appear to "fade into the background" of Metacat's memory. In the present case, the ***wyz*** answer just found is the most strongly activated answer (not surprisingly). As can be seen, this answer reminds the program somewhat of the other ***wyz*** answer. In addition, it also reminds the program of ***uyz***, although to a lesser extent. The other answers, however, lie "too far away" from ***wyz*** to be recalled.[24]

Determining the distances between answers involves essentially the same issues discussed earlier in the context of answer comparison. Distance is a numerical measure computed on the basis of the rules and various types of themes (*i.e.*, common,

[23]Unlike the activations of Slipnet concepts, however, the activations of answer descriptions do not decay over time in the current version of the program. They change only when the program discovers a new answer. This shortcoming should eventually be remedied.

[24]In the current version of the program, it is not possible for snag descriptions to become activated. However, there is no reason in principle why this should be the case. Hitting a snag in one problem could remind the program of similar snags it has encountered in other problems in the same way that it is reminded of other answers—by comparing the description of the just-encountered snag with other snag descriptions in memory.

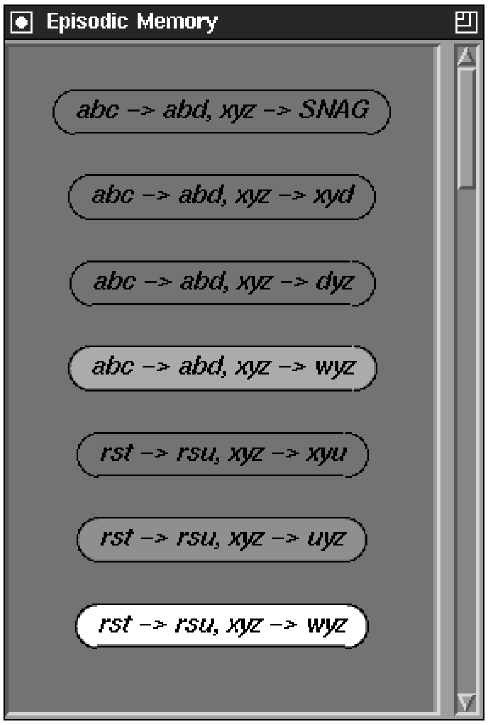


Figure 4.17: *Six answer descriptions (and one snag description) stored in Metacat's Episodic Memory. The answer* ***wyz*** *to the problem "****rst****$\Rightarrow$* ***rsu****;* ***xyz****$\Rightarrow$ ****?****" has just been found, which reminded the program of the same answer to the problem "****abc****$\Rightarrow$* ***abd****;* ***xyz****$\Rightarrow$ ****?****". The program was also reminded, to a lesser extent, of the answer* ***uyz****.*

differing, unique, unjustified, or snag-justified) that exist for a pair of answer descriptions. Specifically, distance is a function of:

1. The number of differing themes and unique themes that exist (the more such themes, the greater the distance between the answers).

2. The number of structural and conceptual differences that exist between the rules involved (the more such differences, the greater the distance).

3. The difference in abstractness of the rules involved (with distance increasing with the degree of disparity).

4. The number of themes that are justified—in one way or another—for one answer but unjustified for the other (the more such themes, the greater the distance).

5. The coherence of the answers (the distance between a coherent and an incoherent answer being greater than the distance between two coherent answers, or two incoherent answers).

One final point deserves to be emphasized. Like Copycat's Slipnet, Metacat's Slipnet serves as the program's ultimate repository for its knowledge of concepts pertaining to the letter-string microworld. These concepts acquire their meanings (*i.e.*, their *semantics*) solely by virtue of the ways in which they can become activated in response to certain situations arising in this world.[25] Accordingly, they represent the "stuff" out of which the program's understanding of its answers—in any genuine sense—arises. Therefore, it is important to emphasize the fact that answer descriptions, which serve as the basis for Metacat's ability to talk about its answers in an insightful manner, are ultimately just organized patterns of Slipnet concepts, since they are composed of themes and rules (which in turn are composed of Slipnet concepts). Thus the English-language commentary generated by the program about its answers, although just a surface-level "gloss" in many ways, nevertheless rests on a deeper foundation of conceptual representation.

[25] See [Hofstadter and FARG, 1995], particularly Chapter 6, for a fuller discussion of this point.

CHAPTER FIVE

Performance of the Model

The previous chapter explained in some detail Metacat's mechanisms that allow it to observe and control its own behavior, and to compare and contrast its answers. This chapter shows these mechanisms in action by presenting a series of annotated runs of the program on a number of different analogy problems. These problems, and their answers, are grouped into several related *families* for the purposes of comparison, and they serve to illustrate, in various ways, Metacat's ability to perceive abstract similarities between answers that, on the surface, would appear to be quite different.

The first section discusses these analogy problems, although most of them have already been discussed to some extent in earlier chapters. This is followed by complete runs of the program, illustrated by screen dumps, showing its behavior on these problems in detail. In a sense, the runs presented here represent Metacat "acting on its best behavior", because they demonstrate the kinds of things that the program is able to do successfully. In contrast, the "darker side" of Metacat is presented in the concluding section—namely, examples that point out a number of serious shortcomings that remain in the current version of the program.

5.1 Three Families of Analogy Problems

5.1.1 The *xyz* family

The first family of analogies involves the pair of problems *"**abc** ⇒ **abd**; **xyz** ⇒ ?"* and *"**rst** ⇒ **rsu**; **xyz** ⇒ ?"* (see the top of Figure 5.1). These problems have been discussed at length already, particularly in section 2.4.6 of Chapter 2. To recap briefly, the answers ***xyd*** and ***xyu*** represent fundamentally identical ways of doing "the same thing" to ***xyz*** as was done to ***abc*** or ***rst***, and are both based on a literal-minded interpretation of the problem. In contrast, a more abstract approach, in which ***xyz*** is viewed as the mirror image of ***abc*** or ***rst***, yields ***wyz*** in each case. However, this really makes sense only for ***abc***, given the lack of ***a***–***z*** symmetry between ***xyz*** and ***rst***. These two analogies, therefore, are quite different in character, even though they both involve exactly the same answer. Finally, a blend of abstract and literal-minded approaches is responsible for the answers ***dyz*** and ***uyz***, making both of these answers seem a bit incoherent. However, it could be argued that since ***abc*** and ***xyz*** are symmetric in every way, while ***rst*** and ***xyz*** are not, changing the ***x*** to ***d*** seems even sillier in the former case than changing it to ***u*** seems in the latter, making ***dyz*** *more* incoherent than ***uyz***. In other words, a subtle but key difference exists between these analogies, on account of the additional alphabetic-position symmetry in the first problem, just as in the case of the two ***wyz*** analogies.

5.1.2 The *mrrjjj* family

The second family of analogies consists of the answers ***mrrkkk*** and ***mrrjjjj*** to the pair of problems *"**abc** ⇒ **abd**; **mrrjjj** ⇒ ?"* and *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ ?"* (see the middle of Figure 5.1). Each of these analogies relies on seeing the target string ***mrrjjj*** in terms of the three components ***m***, ***rr***, and ***jjj***—which correspond to the three letters of the initial string—and on viewing the rightmost letter of the initial

The *xyz* family

$abc \Rightarrow abd$ $xyz \Rightarrow xyd$	$abc \Rightarrow abd$ $xyz \Rightarrow wyz$	$abc \Rightarrow abd$ $xyz \Rightarrow dyz$
$rst \Rightarrow rsu$ $xyz \Rightarrow xyu$	$rst \Rightarrow rsu$ $xyz \Rightarrow wyz$	$rst \Rightarrow rsu$ $xyz \Rightarrow uyz$

The *mrrjjj* family

$abc \Rightarrow abd$ $mrrjjj \Rightarrow mrrkkk$	$abc \Rightarrow abd$ $mrrjjj \Rightarrow mrrjjjj$
$xqc \Rightarrow xqd$ $mrrjjj \Rightarrow mrrkkk$	$xqc \Rightarrow xqd$ $mrrjjj \Rightarrow mrrjjjj$

The *eqe* family

$eqe \Rightarrow qeq$ $abbba \Rightarrow baaab$	$eqe \Rightarrow qeq$ $abbba \Rightarrow aaabaaa$
$eqe \Rightarrow qeq$ $abbbc \Rightarrow qeeeq$	$eqe \Rightarrow qeq$ $abbbc \Rightarrow aaabccc$

Figure 5.1: *Three families of letter-string analogies.*

string as changing to its successor. Accordingly, the rightmost component of ***mrrjjj*** (*i.e.*, the ***jjj*** group) also changes to its successor, yielding either the answer ***mrrkkk*** if ***mrrjjj*** is viewed in terms of *letter-categories* (*i.e.*, as *m–r–j*), or the answer ***mrrjjjj*** if it is viewed in terms of *group-lengths* (*i.e.*, as *1–2–3*).

In the case of 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 this 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 case of *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ ?"*, for similar reasons. That is, unlike ***abc***, the string ***xqc*** possesses no internal structure, so viewing ***mrrjjj*** in an unstructured way, as *m–r–j*, more closely parallels ***xqc*** than does seeing it as *1–2–3*. In a sense then, paying attention to group-lengths in this problem amounts to "being too clever". (Likewise, *not* paying attention to group-lengths in the other problem amounts to "being too obtuse".) The two ***mrrkkk*** answers, therefore, are actually quite different in character, as are the two ***mrrjjjj*** answers.

5.1.3 The *eqe* family

The third family of analogies involves the pair of problems *"**eqe** ⇒ **qeq**; **abbba** ⇒ ?"* and *"**eqe** ⇒ **qeq**; **abbbc** ⇒ ?"* (see the bottom of Figure 5.1). In these two problems (which were discussed earlier in section 4.7.2 of Chapter 4), ***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 consisting of the three components ***a***, ***bbb***, and ***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***. In contrast, this approach won't work for ***abbbc***, because here there are *three* distinct letter-categories involved, hence "swapping" them makes no sense. One way around this difficulty is simply to abandon the idea of swapping altogether, seeing the letters of ***eqe*** as instead changing *individually* to ***q***, ***e***, and ***q***. Changing ***abbbc*** in the analogous way thus amounts to changing its three components, one by one, to ***q***, ***eee***, and ***q***, yielding the answer ***qeeeq***.

A more elegant approach, however, is to reperceive ***abbbc*** 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** ⇒ **?**"* (although in the latter problem, no snag is involved). On the other hand, it is possible to take this approach with ***abbba*** as well, swapping lengths instead of letter-categories to yield ***aaabaaa***. However, as with the earlier answer ***mrrjjjj*** to the problem *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ **?**"*, this amounts to "overkill". That is, there is no good reason to view ***abbba*** as *1–3–1*, since swapping letter-categories works just fine. Thus the difference between the answers ***baaab*** and ***aaabaaa*** to the problem *"**eqe** ⇒ **qeq**; **abbba** ⇒ **?**"* is just like the difference between the answers ***mrrkkk*** and ***mrrjjjj*** to the problem *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ **?**"*, because in both cases viewing the target string in terms of group-lengths—although perhaps seeming like a clever thing to do—actually makes for a *weaker* analogy.

In contrast, viewing the target string in terms of group-lengths in the problems *"**eqe** ⇒ **qeq**; **abbbc** ⇒ **?**"* and *"**abc** ⇒ **abd**; **mrrjjj** ⇒ **?**"* makes for a *stronger* analogy than would otherwise be possible in each case—although not for precisely the same reasons, since doing so in the former problem enables a snag to be avoided, while in the latter problem no snag arises. In other words, the answer ***aaabccc*** to the first problem is a strong answer for both "pragmatic" and aesthetic reasons, while the answer ***mrrjjjj*** to the second problem is a strong answer for aesthetic reasons only. Likewise, the answer ***aaabccc*** to the first problem represents a stronger analogy than the answer ***aaabaaa*** to the problem *"**eqe** ⇒ **qeq**; **abbba** ⇒ **?**"*, even though both analogies involve seeing the target string as *1–3–1*, precisely because of the fact that paying attention to group-lengths is warranted in the former case (due to the potential for a snag) but not in the latter.

5.2 Sample Runs of the Program

This section presents a selection of sample runs of Metacat on several of the above problems, in order to illustrate more clearly the mechanisms discussed in Chapter 4. For each of these runs, a series of "snapshots" of the Temporal Trace and Workspace is shown, giving a sense of how the run evolves over time. These snapshots, taken directly from the screen, are intended to highlight the most interesting aspects of a run. Unfortunately, though, they do not show the many different colors that are used to color-code various structures for clarity, especially in the Workspace. However, one particular feature of the program can be used to compensate for this, at least to some degree. In general, every event appearing in the Temporal Trace can be *displayed* by clicking on its graphical icon with the mouse. When this is done, the structures associated with the event are highlighted against a faded grey background consisting of the Workspace structures that existed at the time the event occurred. (In addition, the event's icon is also highlighted in the Trace.) This method of displaying events helps to improve the clarity of the examples, and is thus used whenever possible in the following runs.

5.2.1 Examples of answer justification

Run 1: *abc* ⇒ *abd; mrrjjj* ⇒ *mrrjjjj*

The first run demonstrates Metacat's ability to justify the answer ***mrrjjjj*** to the problem "***abc*** ⇒ ***abd; mrrjjj*** ⇒ ***?***" through the application of top-down pressure generated by clamping patterns. The run begins in the usual way, with codelets building bonds, groups, bridges, and other Workspace structures among the letter-strings. By about 500 codelets[1], sameness groups have been built in both ***mrrjjj*** and ***mrrjjjj***, but their lengths have not yet been noticed by the program. (A *Length* description has been attached to the ***rr*** group in ***mrrjjjj***, but this description is irrelevant, because the *Length* concept in the Slipnet is not activated.) In addition, the activation of the concept of *identity* has been recorded in the Trace, as a result of the creation of bridges involving identity concept-mappings such as *leftmost* ⇒ *leftmost* and *letter* ⇒ *letter* (see Panel *1-a*).

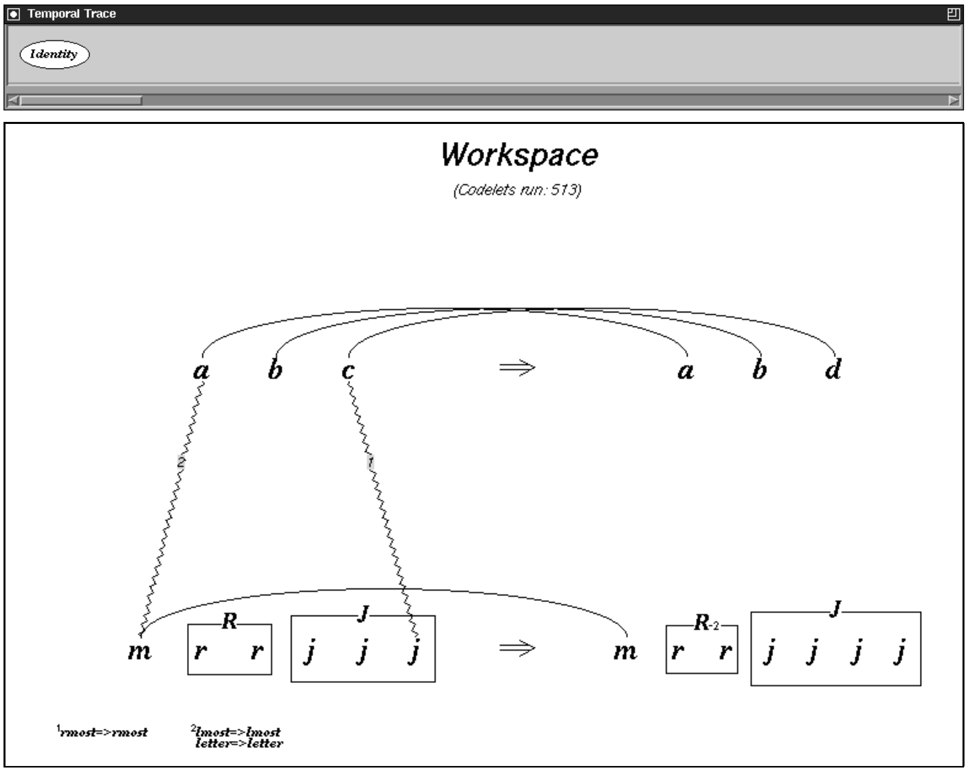
Soon after this, the top rule *Change letter-category of rightmost letter to successor* is built, based on the horizontal bridges between ***abc*** and ***abd***. The next important event occurs when the program perceives the letter ***m*** in ***mrrjjj*** as a sameness group of length one, due to the top-down pressure exerted by the *sameness-group* concept in the Slipnet, which has become activated by the presence of the other sameness groups in ***mrrjjj*** and ***mrrjjjj*** (see Panel *1-b*).

Over the course of the next 500 or so time steps, the ***m*** in ***mrrjjjj*** is likewise perceived as a single-letter group, the entire initial string is perceived as a successor group, and another top rule is created for describing ***abc*** ⇒ ***abd*** (*Change letter-category of letter 'c' to 'd'*). At time step 1187, the bottom rule *Increase length of rightmost group by one*[2] gets built, paving the way for unification to occur with the original top rule *Change letter-category of rightmost letter to successor* (see Panel *1-c*). Accordingly, when an *Answer-justifier* codelet runs a few time steps later, it picks the bottom rule and translates it as *Increase length of rightmost letter by one*, a rule which neither exists nor works for ***abc***. Upon examining the existing top rules, however, the codelet discovers that the bottom rule can be unified with the first top rule on the

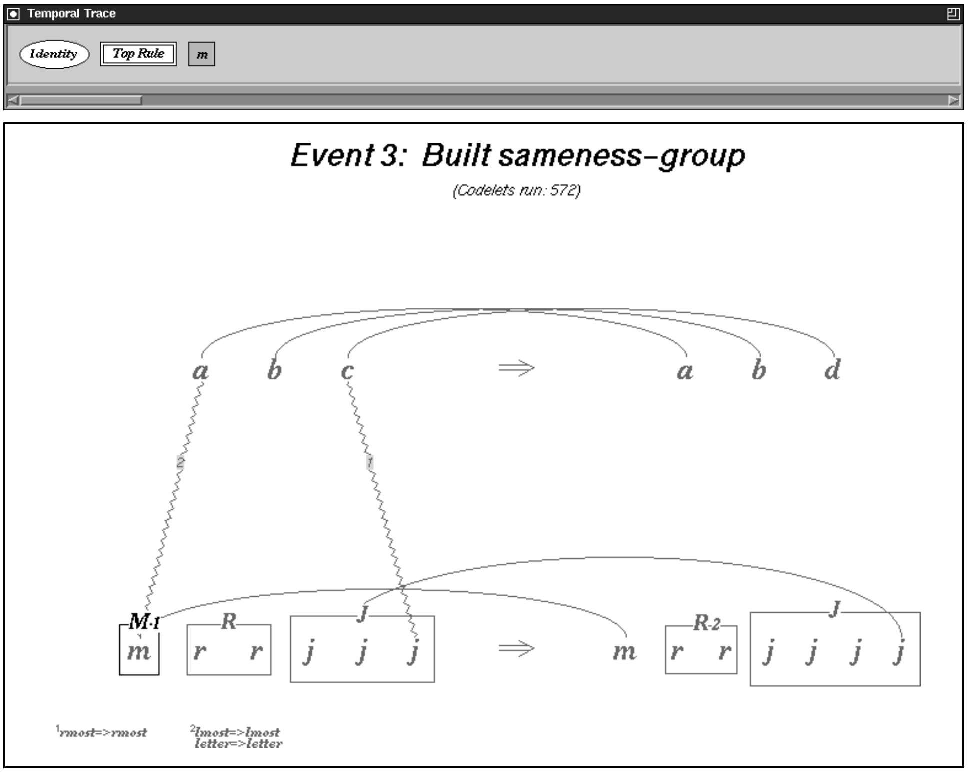
[1]Generally speaking, more codelets are required for a typical run of Metacat than for a typical run of Copycat, since in Metacat, bonds and groups can be built in all of the strings, and horizontal bridges can be built between the top strings (as well as between the bottom strings when running in justify mode).

[2]Or, equivalently, *Change length of rightmost group to successor.*

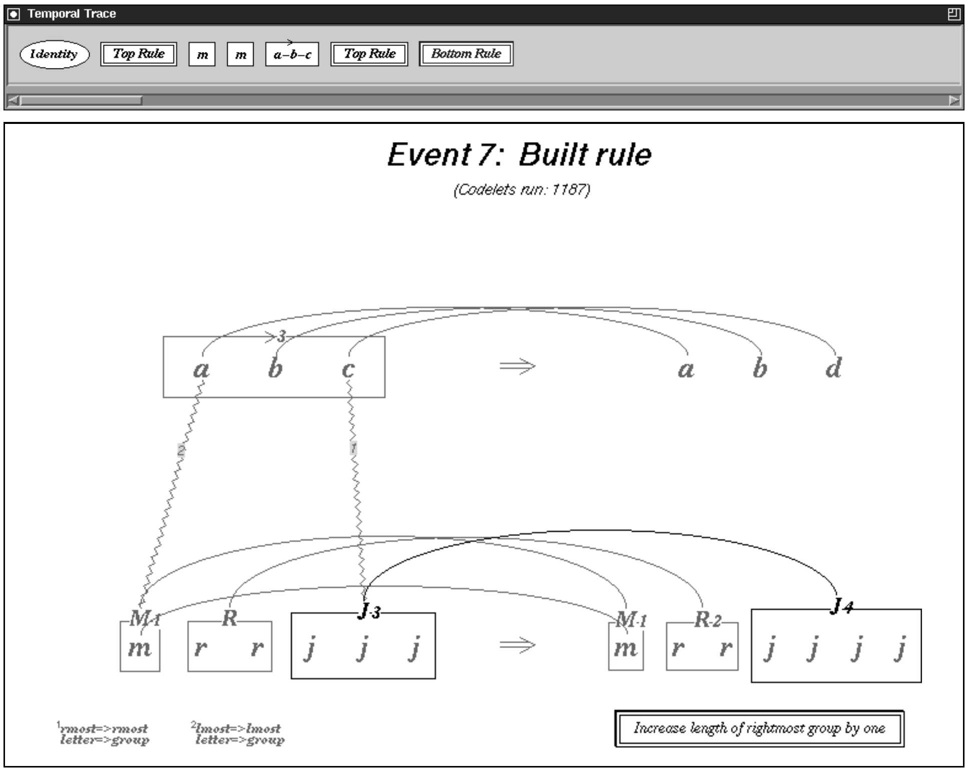
1-a


Temporal Trace
Identity
Workspace
(Codelets run: 513)
a b c ⇒ a b d
m R r r J j j j ⇒ m R-2 r r J j j j j
1rmost=>rmost
2lmost=>lmost letter=>letter

1-b


Temporal Trace
Identity
Top Rule
m
Event 3: Built sameness-group
(Codelets run: 572)
a b c ⇒ a b d
M-1 m R r r J j j j ⇒ m R-2 r r J j j j j
1rmost=>rmost
2lmost=>lmost letter=>letter

1-c

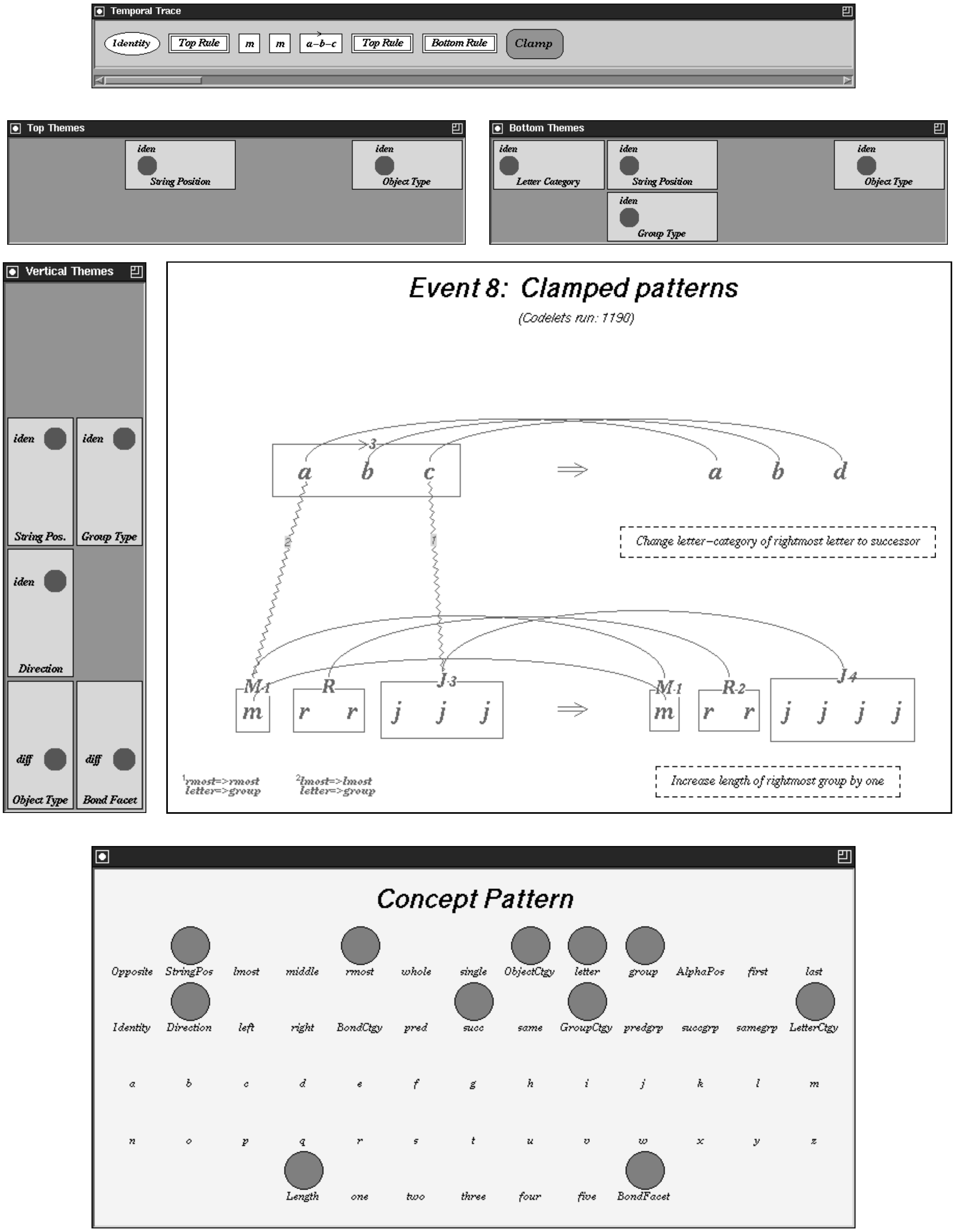


basis of the slippages *Length* ⇒ *Letter-Category* and *group* ⇒ *letter*.[3] (The codelet discovers this by probabilistically picking a top rule of roughly the same strength as the bottom rule, and then comparing its internal structure to that of the bottom rule.) Consequently, it creates a vertical theme-pattern based on these slippages (and on the identity concept-mappings *rightmost* ⇒ *rightmost* and *successor* ⇒ *successor*), which it then clamps in the Themespace, along with the theme-patterns associated with each of the rules. In addition, it clamps a concept-pattern in the Slipnet consisting of concepts associated with the rules and themes.[4] These patterns are shown in Panel *1-d*. In the case of the vertical theme-pattern, the three identity themes arise from the concept-mappings *rightmost* ⇒ *rightmost* and *successor* ⇒ *successor*,

[3]As this example shows, slippages that move *upwards* from the bottom situation to the top situation are possible when Metacat runs in justify mode, contrary to the usual top-to-bottom slippages that arise when the program looks for answers on its own.

[4]This pattern is actually a composite of the individual concept-patterns associated with each of the rules and each of the theme-patterns.

1-d

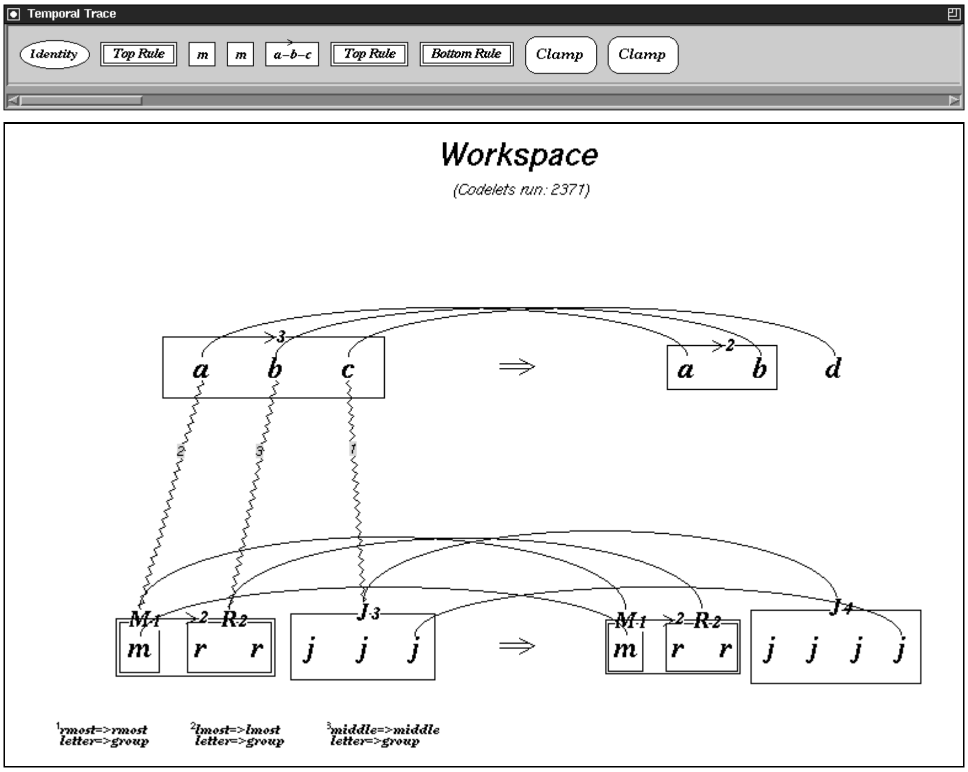

Temporal Trace
Identity
Top Rule
m
m
a–b–c
Top Rule
Bottom Rule
Clamp
Top Themes
iden
String Position
iden
Object Type
Bottom Themes
iden
Letter Category
iden
String Position
iden
Group Type
iden
Object Type
Vertical Themes
iden
String Pos.
iden
Group Type
iden
Direction
diff
Object Type
diff
Bond Facet
Event 8: Clamped patterns
(Codelets run: 1190)
a b c ⇒ a b d
Change letter–category of rightmost letter to successor
m r r j j j ⇒ m r r j j j j
Increase length of rightmost group by one
1 rmost=>rmost letter=>group
2 lmost=>lmost letter=>group
Concept Pattern
Opposite
StringPos
lmost
middle
rmost
whole
single
ObjectCtgy
letter
group
AlphaPos
first
last
Identity
Direction
left
right
BondCtgy
pred
succ
same
GroupCtgy
predgrp
succgrp
samegrp
LetterCtgy
a b c d e f g h i j k l m
n o p q r s t u v w x y z
Length
one
two
three
four
five
BondFacet

while the *Object-Type: different* and *Bond-Facet: different* themes arise from the slippages *group* ⇒ *letter* and *Length* ⇒ *Letter-Category*. The top and bottom theme-patterns reflect the identity concept-mappings underlying the top and bottom horizontal bridges. Additionally, a codelet-pattern is clamped that enhances the urgencies of top-down bond, group, and description codelets. This codelet-pattern (not shown in the figure) works in tandem with the clamped concept-pattern to promote the creation of other types of Workspace structures besides bridges.

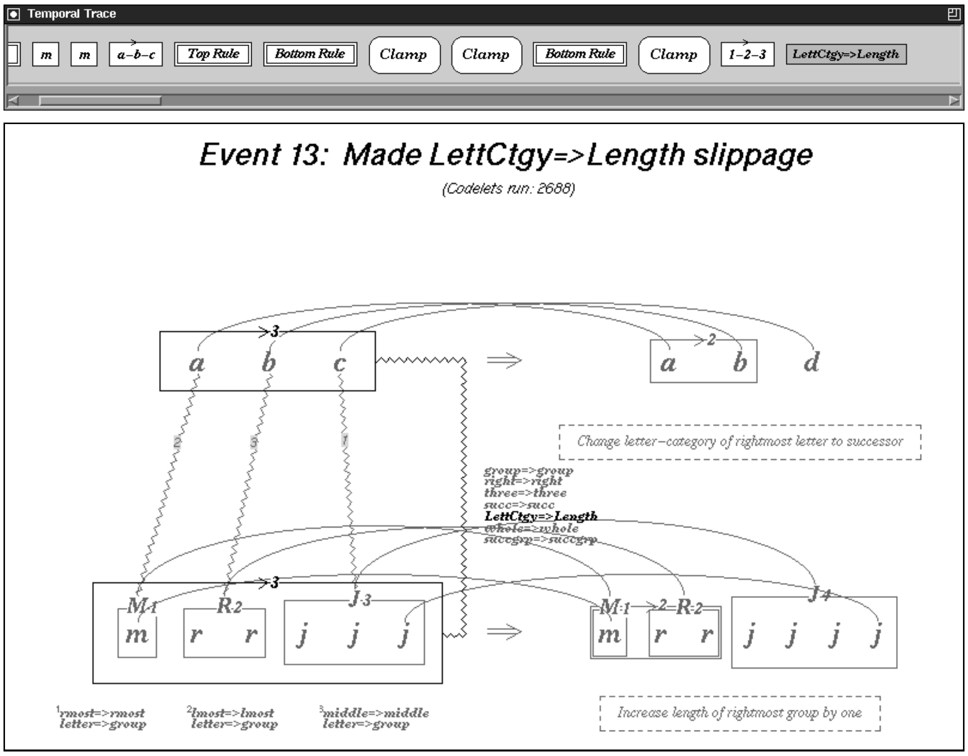
The effect of the resulting top-down pressure is twofold. First, the clamped theme-patterns essentially hold the existing string mappings in place, since none of the bridges are incompatible with the clamped themes. Second, the clamped concept- and codelet-patterns increase the likelihood that a *Length* description will get attached to the ***rr*** group in ***mrrjjj***, on account of the clamped *Length* concept and the enhanced urgencies of *Description-scout* codelets. In fact, this occurs approximately 250 time steps later—still well within the clamp period. Furthermore, once this description has been attached to ***rr***, ***mrrjjj*** is more likely to get chunked into a high-level successor group on account of the clamped *successor* and *Length* concepts in the Slipnet and the enhanced urgencies of *Bond-scout* and *Group-scout* codelets.

As it turns out, the clamp period ends before this happens. Approximately 200 codelets later, at time step 1826, the program again tries to unify the same pair of rules as before. This effort again results in a justify clamp, because no bridge yet exists between ***abc*** and ***mrrjjj*** as a whole. By the end of the second clamp period at time step 2371, the program has perceived ***mrr*** as a *1–2* successor group in both ***mrrjjj*** and ***mrrjjjj***, but as yet no whole-string group exists (see Panel *1-e*). By time step 2506, the program has managed only to create another bottom rule, *Change length of rightmost group to four*, which doesn't help the situation very much. A third justify clamp thus ensues, based on the same pair of rules as before. This time, however, the *1–2–3* successor group gets created approximately 130 codelets into the clamp period.

1-e


Temporal Trace
Identity
Top Rule
m
m
a–b–c
Top Rule
Bottom Rule
Clamp
Clamp
Workspace
(Codelets run: 2371)
a b c ⇒ a b d
m r r j j j ⇒ m r r j j j j
[1]rmost=>rmost letter=>group
[2]lmost=>lmost letter=>group
[3]middle=>middle letter=>group

1-f


Temporal Trace
m
m
a–b–c
Top Rule
Bottom Rule
Clamp
Clamp
Bottom Rule
Clamp
1–2–3
LettCtgy=>Length
Event 13: Made LettCtgy=>Length slippage
(Codelets run: 2688)
a b c ⇒ a b d
Change letter–category of rightmost letter to successor
group=>group
right=>right
three=>three
succ=>succ
LettCtgy=>Length
whole=>whole
succgrp=>succgrp
m r r j j j ⇒ m r r j j j j
[1]rmost=>rmost letter=>group
[2]lmost=>lmost letter=>group
[3]middle=>middle letter=>group
Increase length of rightmost group by one

Soon afterwards, still under pressure from the vertical theme-pattern, a bridge is built between the two whole-string successor groups ***abc*** and ***mrrjjj*** at time step 2688, based in part on the slippage *Letter-Category* ⇒ *Length*. Because this slippage is compatible with the clamped *Bond-Facet: different* theme, it is considered important enough to be recorded in the Trace (see Panel *1-f*).

Finally, just a few time steps later, an *Answer-justifier* codelet runs, this time successfully unifying the rules on the basis of the existing vertical mapping. A coherent interpretation of the problem yielding the answer ***mrrjjjj*** has thus been discovered. At this point, the codelet creates an abstract description of the answer (shown in Figure 5.2) from the themes associated with the slippage event in the Trace, the

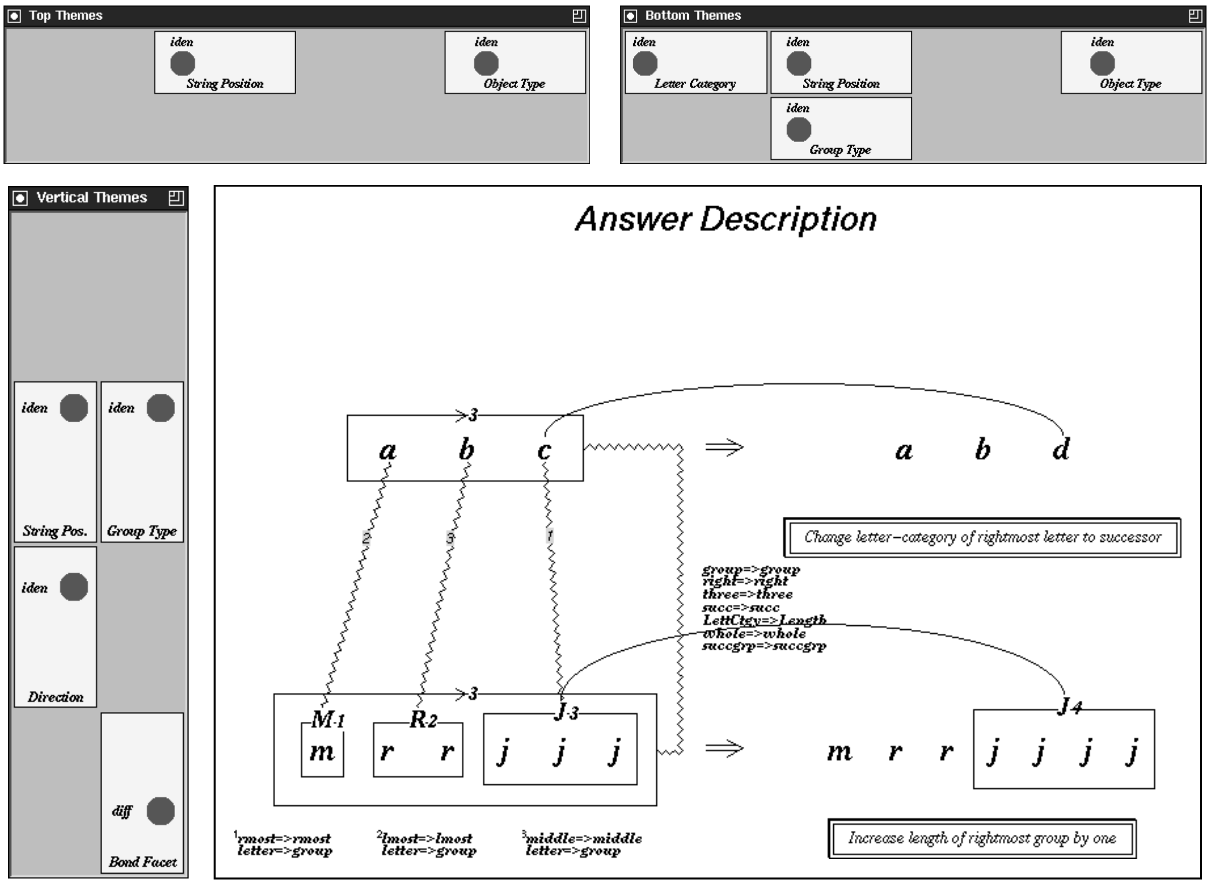


Figure 5.2: *The answer description for* ***mrrjjjj*** *created at the end of Run 1.*

whole-string bridge in the Workspace, and the top and bottom rules, which it then stores in the Episodic Memory.

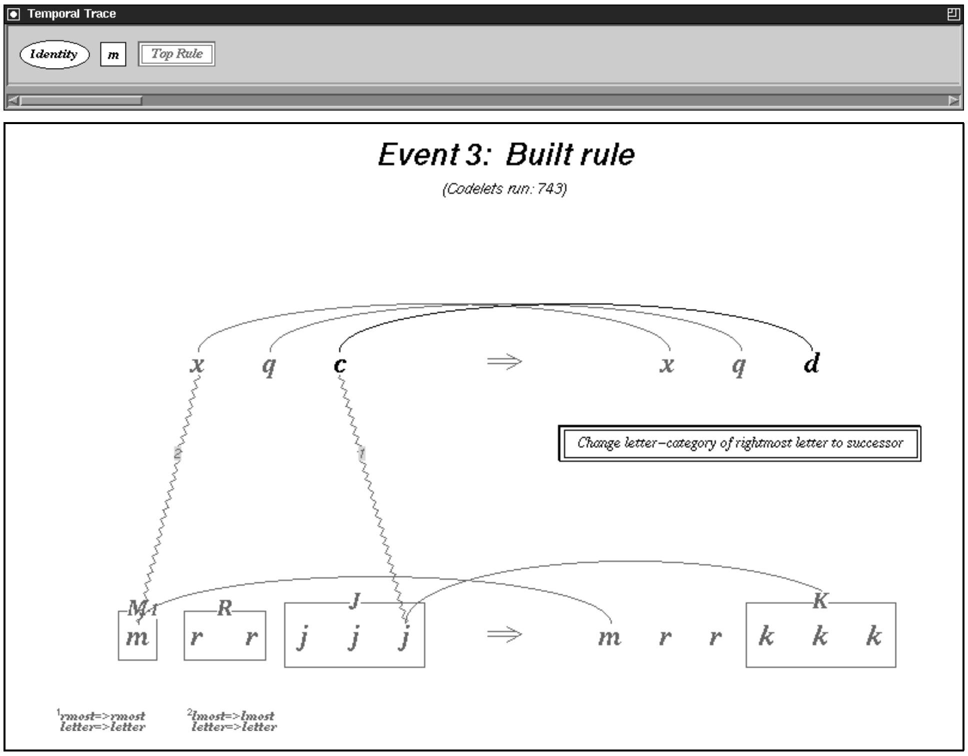
Run 2: *xqc* ⇒ *xqd; mrrjjj* ⇒ *mrrkkk*

In Run 1, the clamped theme-patterns serve to lock in the existing string mappings so that they will not be inadvertently destroyed while the program is waiting for other types of structures to be built, such as descriptions and groups. By contrast, Run 2 illustrates the use of thematic pressure to reorganize a mapping in support of a particular rule. In this run, the program is given the answer ***mrrkkk*** to the problem *"**xqc** ⇒ **xqd**; **mrrjjj** ⇒ **?**"* and asked to justify it. Because this answer does not require seeing ***mrrjjj*** as a single group, it is easier for the program to justify than the answer ***mrrjjjj*** in Run 1.

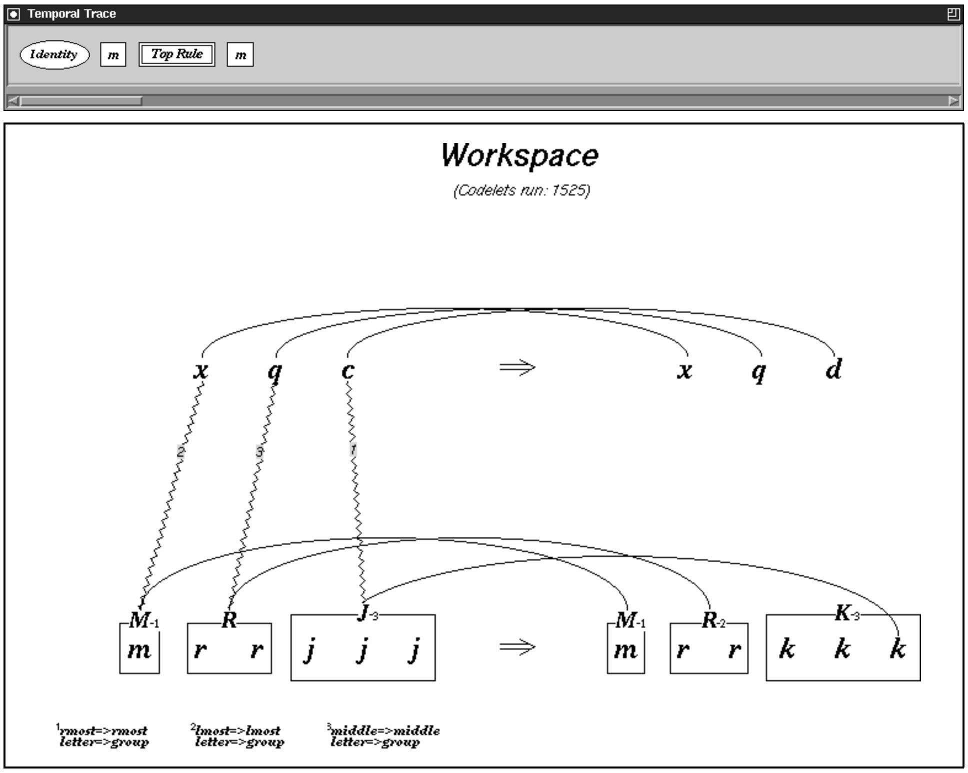
The beginning of the run is similar to that of Run 1. At time step 743, the program builds the top rule *Change letter-category of rightmost letter to successor* (see Panel *2-a*). By time step 1525, the program has built up strong top and vertical string mappings, but the bottom mapping is not quite uniform, since the ***jjj*** group of ***mrrjjj*** is seen as corresponding to the rightmost *letter* ***k*** of ***mrrkkk*** (see Panel *2-b*). Shortly thereafter, at time step 1550, an *Answer-justifier* codelet runs, picking the top rule and translating it as *Change letter-category of rightmost group to successor.* As it turns out, this translated rule works correctly for ***mrrjjj***, so the codelet adds it to the Workspace as a new bottom rule, since no such rule yet exists. Unfortunately, however, this new rule is not currently supported by the mapping between ***mrrjjj*** and ***mrrkkk***, on account of the ***jjj***–***k*** bridge (see Panel *2-c*).[5] Consequently, the codelet clamps the theme-patterns associated with each of the rules, together with

[5]The darker ***jjj***–***kkk*** bridge shown in the panel does not yet exist in the Workspace. Existing Workspace structures are shown in grey. In general, whenever a rule event is displayed by the user, the bridges required to support the rule are also shown, whether or not they currently exist. This brings out more clearly any inconsistencies with the current set of Workspace bridges.

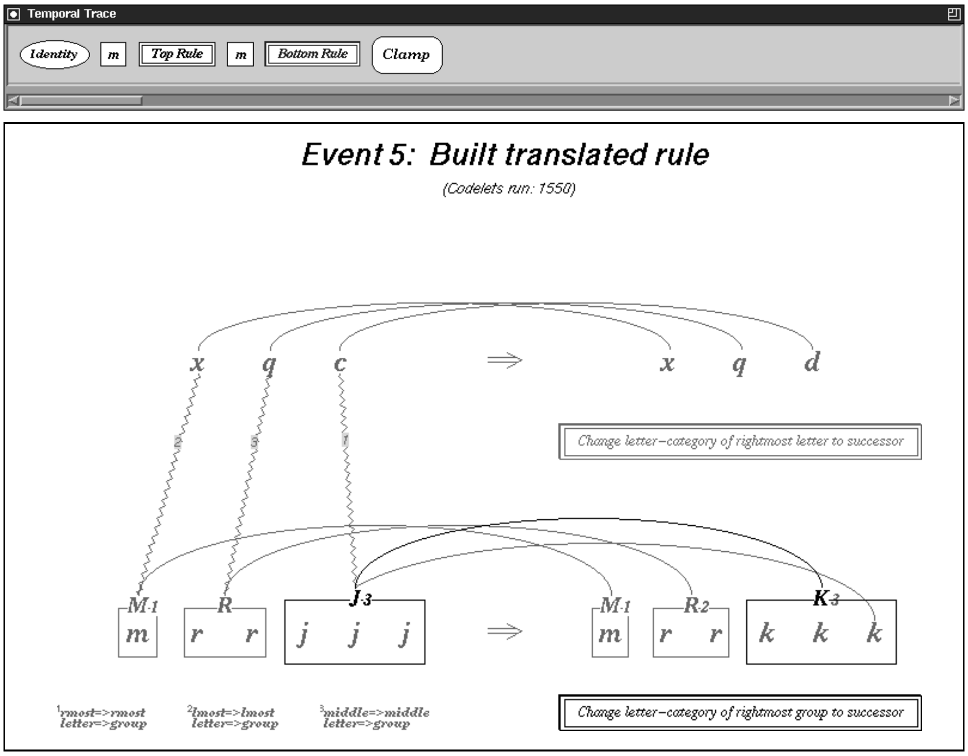
2-a


Temporal Trace
Identity
m
Top Rule
Event 3: Built rule
(Codelets run: 743)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M R J K
m r r j j j ⇒ m r r k k k
1rmost=>rmost letter=>letter
2lmost=>lmost letter=>letter

2-b

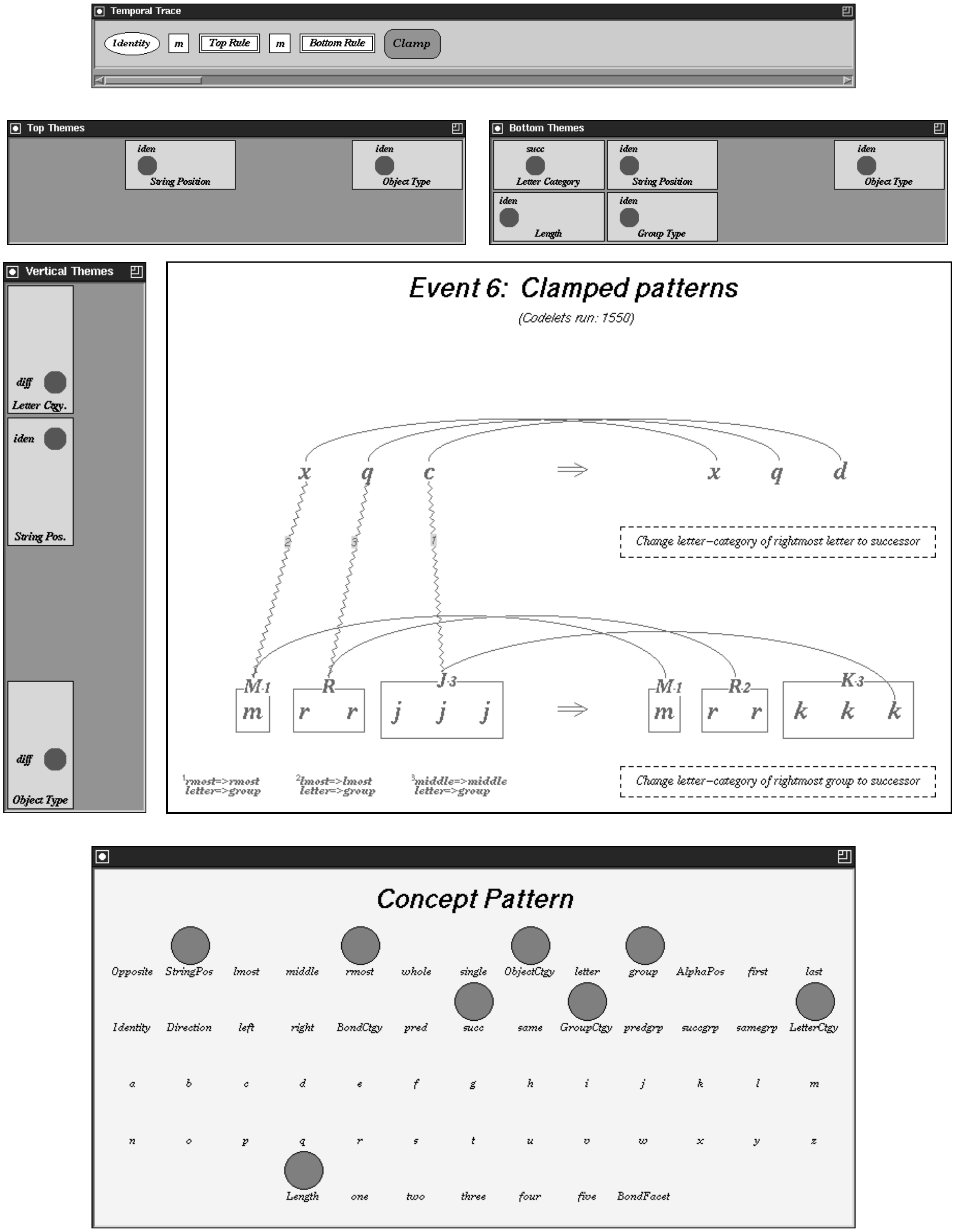

Temporal Trace
Identity
m
Top Rule
m
Workspace
(Codelets run: 1525)
x q c ⇒ x q d
M R J M R K
m r r j j j ⇒ m r r k k k
1rmost=>rmost letter=>group
2lmost=>lmost letter=>group
3middle=>middle letter=>group

2-c

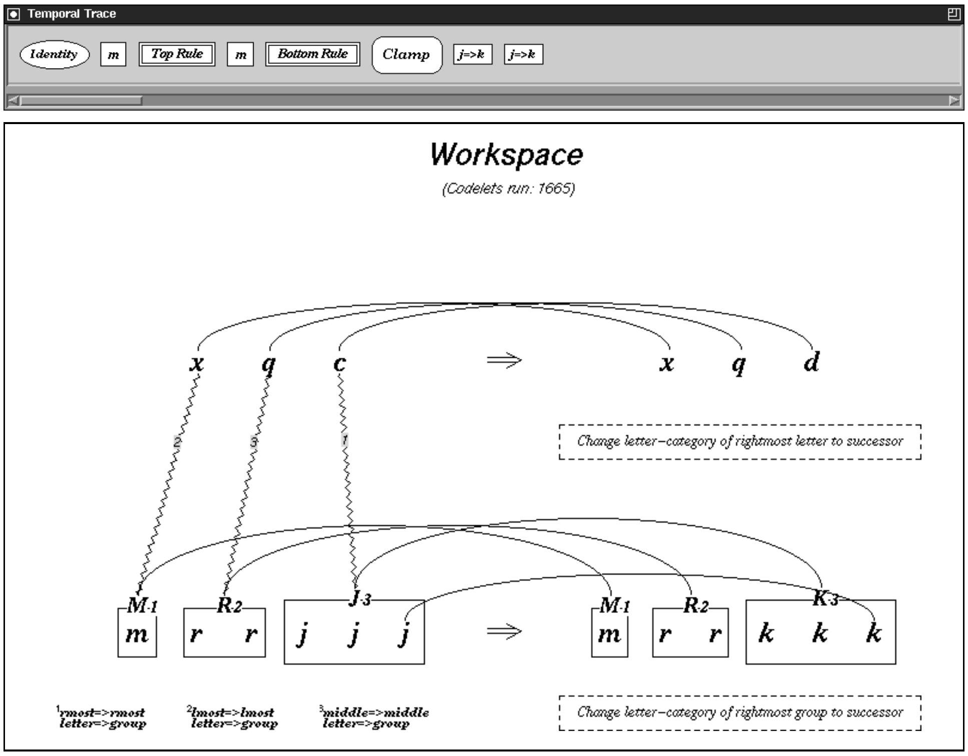


the current pattern of *dominant* vertical themes in the Themespace (see Panel *2-d*). Clamping the bottom theme-pattern greatly weakens the existing ***jjj–k*** bridge, due to the incompatible *Object-Type: identity* theme, and strongly promotes the creation of a bridge between the two groups ***jjj*** and ***kkk***. All other existing bridges are compatible with the clamped themes, so their strengths get reinforced by the clamp. The net effect is that the bottom mapping gets "cleaned up" while the other two mappings are held in place. By time step 1665, approximately 100 codelets into the clamp period, the bottom mapping is consistent with the bottom rule (see Panel *2-e*). The two $j \Rightarrow k$ slippage events appearing in the Trace are associated with the newly-created bridges ***jjj–kkk*** and ***j–k***, each of which is compatible with the clamped bottom themes. This new mapping supports the bottom rule, which in turn paves the way for a successful justification of ***mrrkkk*** at time step 1747 (see Panel *2-f*).

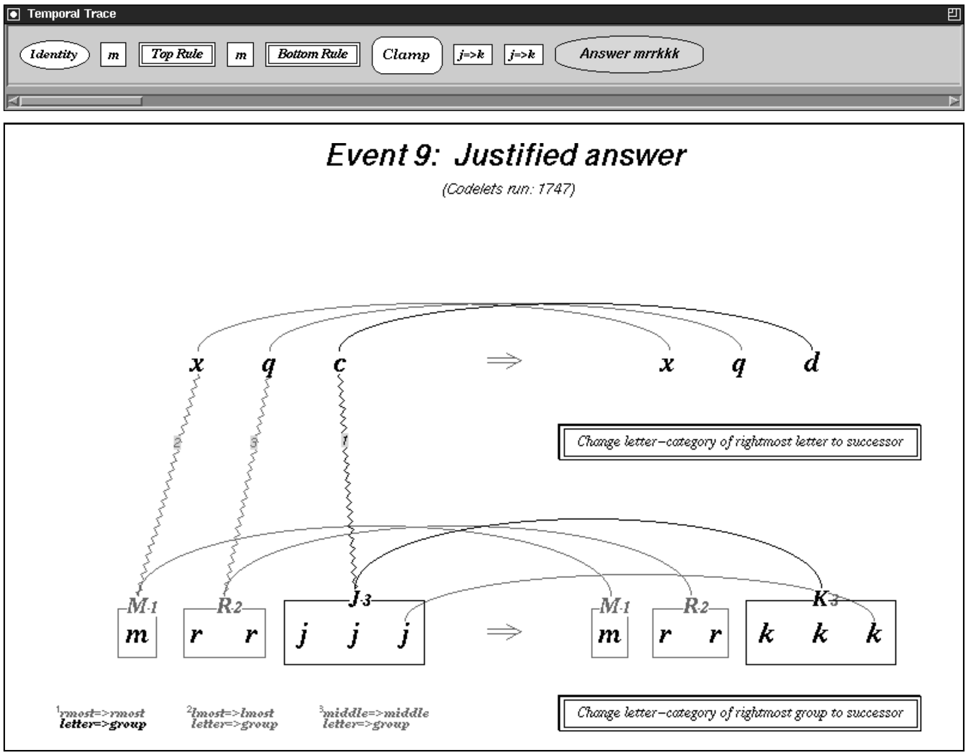
2-d


Temporal Trace
Identity
m
Top Rule
m
Bottom Rule
Clamp
Top Themes
iden
String Position
iden
Object Type
Bottom Themes
succ
Letter Category
iden
String Position
iden
Object Type
iden
Length
iden
Group Type
Vertical Themes
diff
Letter Ctgy.
iden
String Pos.
diff
Object Type
Event 6: Clamped patterns
(Codelets run: 1550)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M·1 R J·3 ⇒ M·1 R·2 K·3
m r r j j j ⇒ m r r k k k
1 rmost=>rmost letter=>group
2 lmost=>lmost letter=>group
3 middle=>middle letter=>group
Change letter–category of rightmost group to successor
Concept Pattern
Opposite StringPos lmost middle rmost whole single ObjectCtgy letter group AlphaPos first last
Identity Direction left right BondCtgy pred succ same GroupCtgy predgrp succgrp samegrp LetterCtgy
a b c d e f g h i j k l m
n o p q r s t u v w x y z
Length one two three four five BondFacet

2-e


Temporal Trace
Identity
m
Top Rule
m
Bottom Rule
Clamp
j=>k
j=>k
Workspace
(Codelets run: 1665)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M1 R2 J3
m r r j j j ⇒ m r r k k k
M1 R2 K3
1rmost=>rmost letter=>group
2lmost=>lmost letter=>group
3middle=>middle letter=>group
Change letter–category of rightmost group to successor

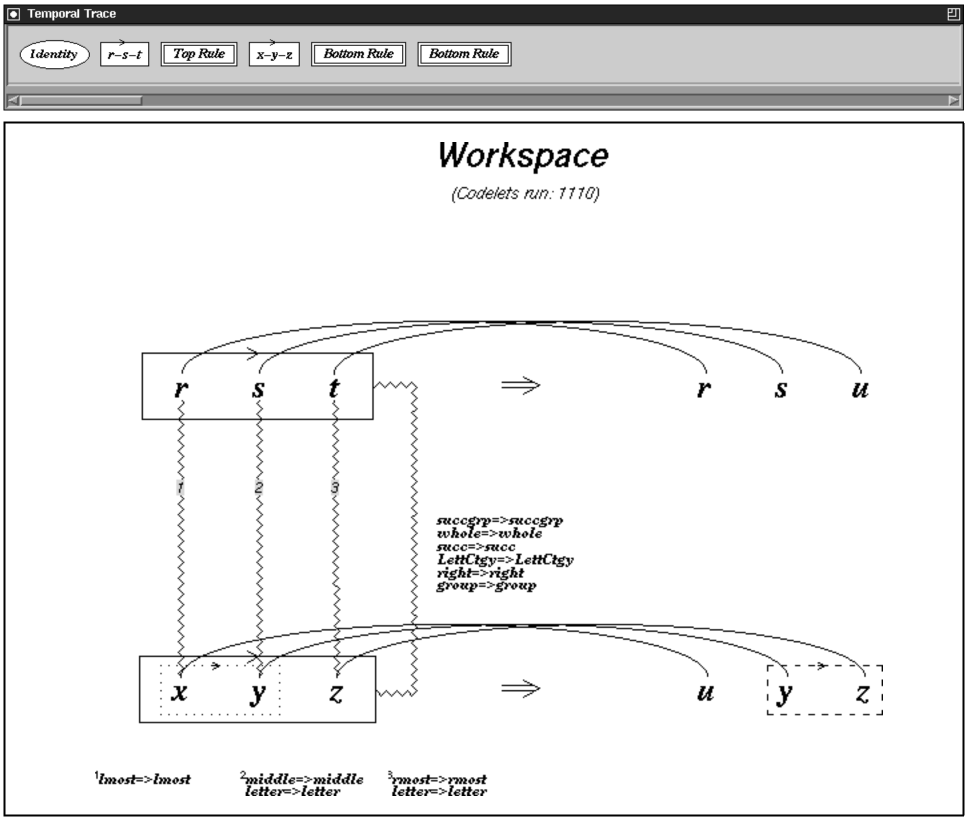
2-f


Temporal Trace
Identity
m
Top Rule
m
Bottom Rule
Clamp
j=>k
j=>k
Answer mrrkkk
Event 9: Justified answer
(Codelets run: 1747)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M1 R2 J3
m r r j j j ⇒ m r r k k k
M1 R2 K3
1rmost=>rmost letter=>group
2lmost=>lmost letter=>group
3middle=>middle letter=>group
Change letter–category of rightmost group to successor

Run 3: *rst* ⇒ *rsu*; *xyz* ⇒ *uyz*

The next example demonstrates the program's ability to spur the creation of new rules by clamping codelet-patterns. For this run, the program is asked to justify the answer ***uyz*** to the problem *"**rst** ⇒ **rsu**; **xyz** ⇒ **?**"*. This answer hinges on mapping ***rst*** and ***xyz*** onto each other in a crosswise fashion, and on viewing both ***t*** and ***x*** as changing literally to ***u***. Initially, however, the program sees ***rst*** and ***xyz*** as going in the same direction. By time step 850, it has built a strong same-direction mapping between these strings, and has created the top rule *Change letter-category of rightmost letter to successor* to describe the ***rst*** ⇒ ***rsu*** change. The two bottom rules *Change letter-category of leftmost letter to 'u'* and *Change letter-category of letter 'x' to 'u'*, describing the ***xyz*** ⇒ ***uyz*** change, are created at time steps 949 and 1101 (see Panel *3-a*).

3-a

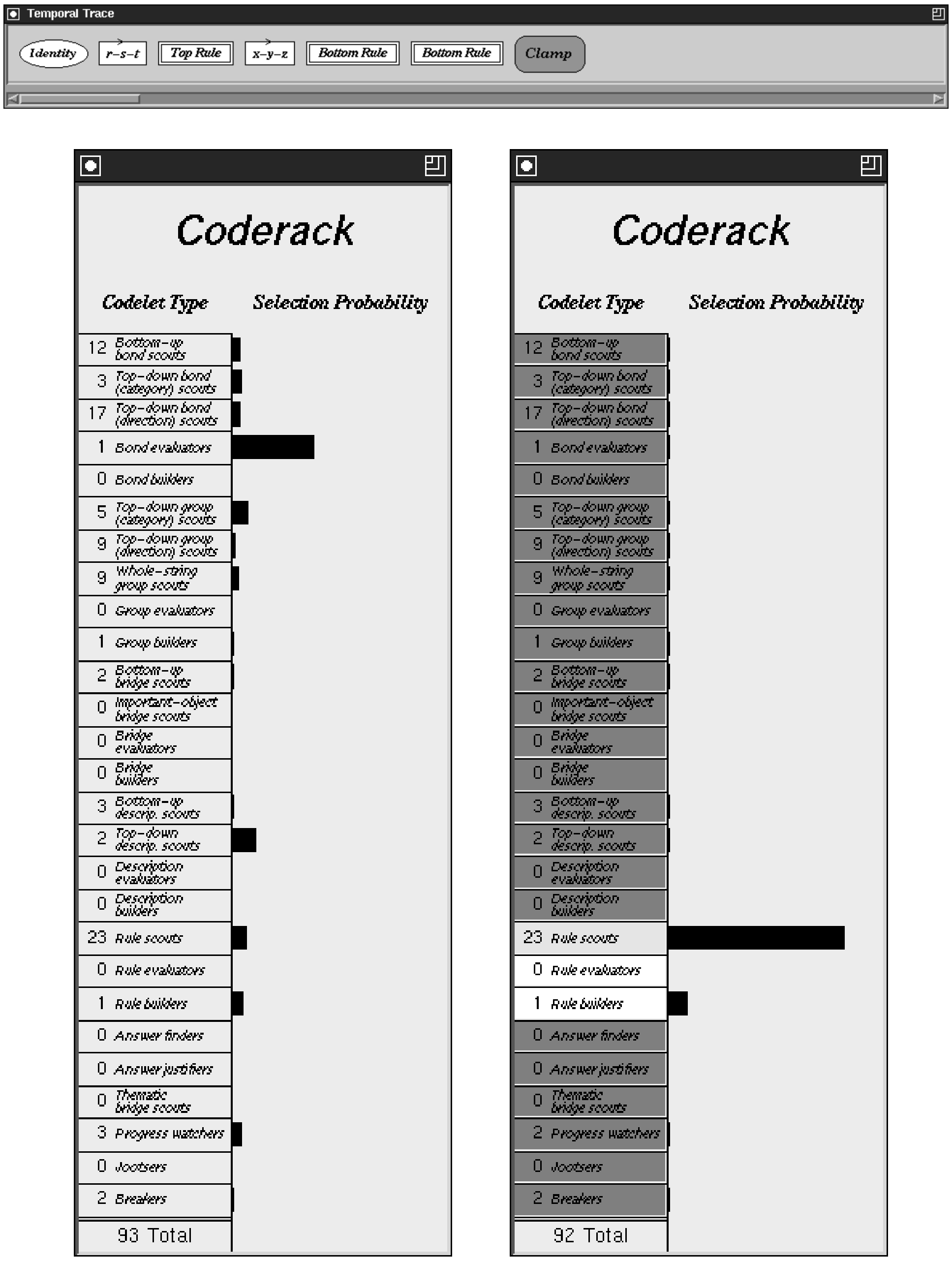


After this, however, the program "runs out of steam", because the existing rules and string mappings do not support the answer ***uyz***, and neither bottom rule can be unified with the top rule. Almost 400 time steps later, the situation has not changed, prompting the program to comment that it is "frustrated" (since by now the Workspace activity has dropped to zero). More precisely, at time step 1495, a *Progress-watcher* codelet notices the lack of Workspace activity and consequently examines the quality of the rules that have been created so far. As it happens, both bottom rules are of low quality, since they both describe the ***xyz*** ⇒ ***uyz*** change in a literal rather than an abstract way.[6] The codelet therefore decides to accelerate the process of rule discovery by clamping the codelet-pattern shown in Panel *3-b* (the left image shows the Coderack just before the *Progress-watcher* codelet runs). In a sense, however, the codelet makes the right decision for the wrong reason, because the current impasse is due not to a lack of good bottom rules, but rather to the absence of a top rule that can be unified with one of the existing bottom rules.

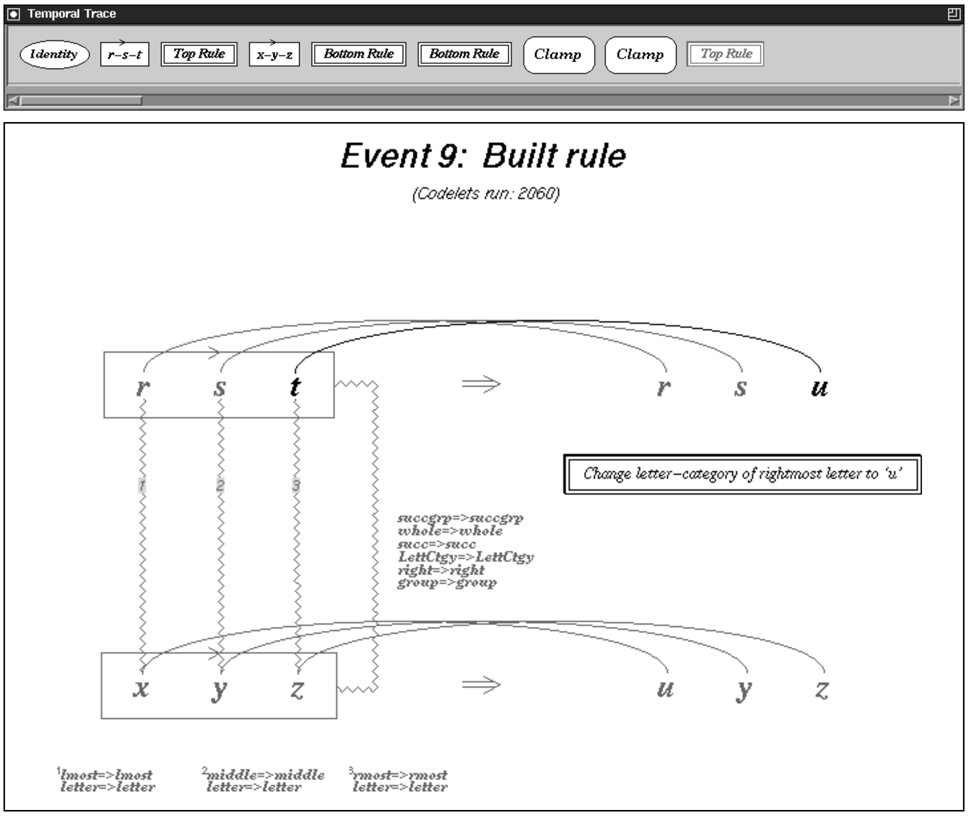
As it turns out, the clamp period ends before any new rules are created, resulting in zero progress achieved by the clamp. At time step 2000, however, the program tries again. This time, the top rule *Change letter-category of rightmost letter to 'u'* gets built just 60 codelets into the clamp period (see Panel *3-c*), paving the way for unification to occur with the bottom rule *Change letter-category of leftmost letter to 'u'*. The latter event happens at time 2656, and results in the clamping of the vertical themes *String-Position: opposite* and *Direction: opposite*, on account of the *rightmost* ⇒ *leftmost* slippage arising from unification, which immediately activates the concept of *opposite* in the Slipnet (see Panel *3-d*). The reinterpretation of ***rst*** as a left-directed predecessor group quickly follows, leading to a successful justification of the answer ***uyz*** at time 3163 (see Panels *3-e* and *3-f*).

[6] Of course, the program is incapable of describing this change in a more abstract way, but it does not know this about itself.

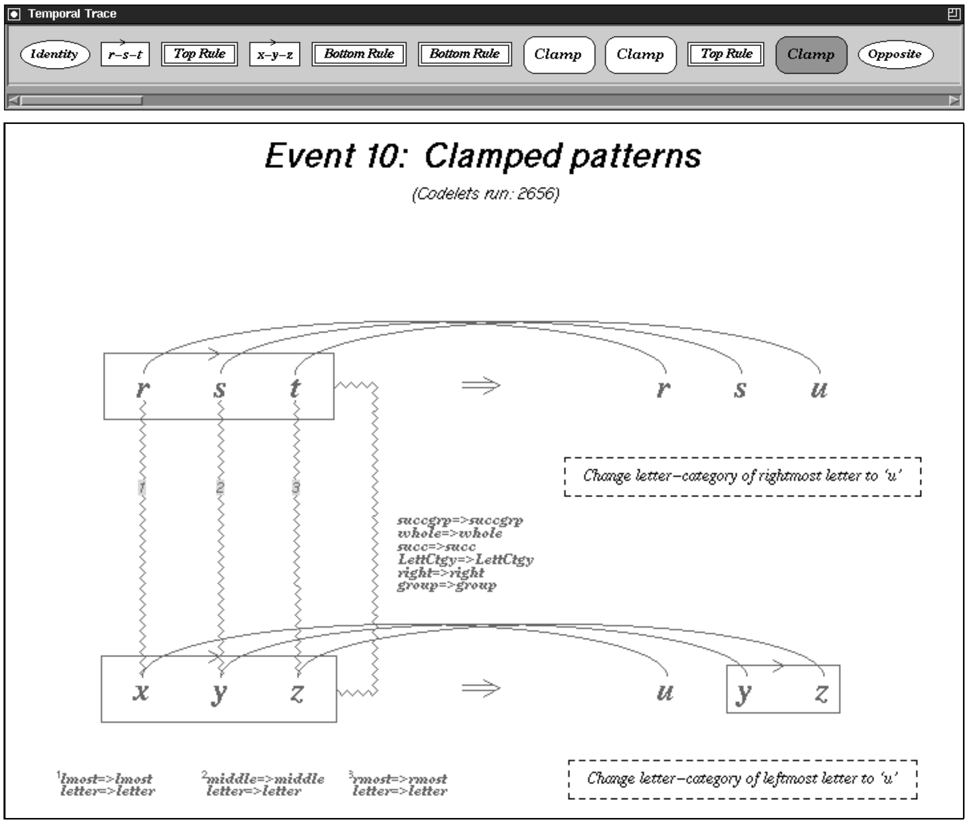
3-b


Temporal Trace
Identity
r–s–t
Top Rule
x–y–z
Bottom Rule
Bottom Rule
Clamp
Coderack
Codelet Type
Selection Probability
12 Bottom–up bond scouts
3 Top–down bond (category) scouts
17 Top–down bond (direction) scouts
1 Bond evaluators
0 Bond builders
5 Top–down group (category) scouts
9 Top–down group (direction) scouts
9 Whole–string group scouts
0 Group evaluators
1 Group builders
2 Bottom–up bridge scouts
0 Important–object bridge scouts
0 Bridge evaluators
0 Bridge builders
3 Bottom–up descrip. scouts
2 Top–down descrip. scouts
0 Description evaluators
0 Description builders
23 Rule scouts
0 Rule evaluators
1 Rule builders
0 Answer finders
0 Answer justifiers
0 Thematic bridge scouts
3 Progress watchers
0 Jootsers
2 Breakers
93 Total
Coderack
Codelet Type
Selection Probability
12 Bottom–up bond scouts
3 Top–down bond (category) scouts
17 Top–down bond (direction) scouts
1 Bond evaluators
0 Bond builders
5 Top–down group (category) scouts
9 Top–down group (direction) scouts
9 Whole–string group scouts
0 Group evaluators
1 Group builders
2 Bottom–up bridge scouts
0 Important–object bridge scouts
0 Bridge evaluators
0 Bridge builders
3 Bottom–up descrip. scouts
2 Top–down descrip. scouts
0 Description evaluators
0 Description builders
23 Rule scouts
0 Rule evaluators
1 Rule builders
0 Answer finders
0 Answer justifiers
0 Thematic bridge scouts
2 Progress watchers
0 Jootsers
2 Breakers
92 Total

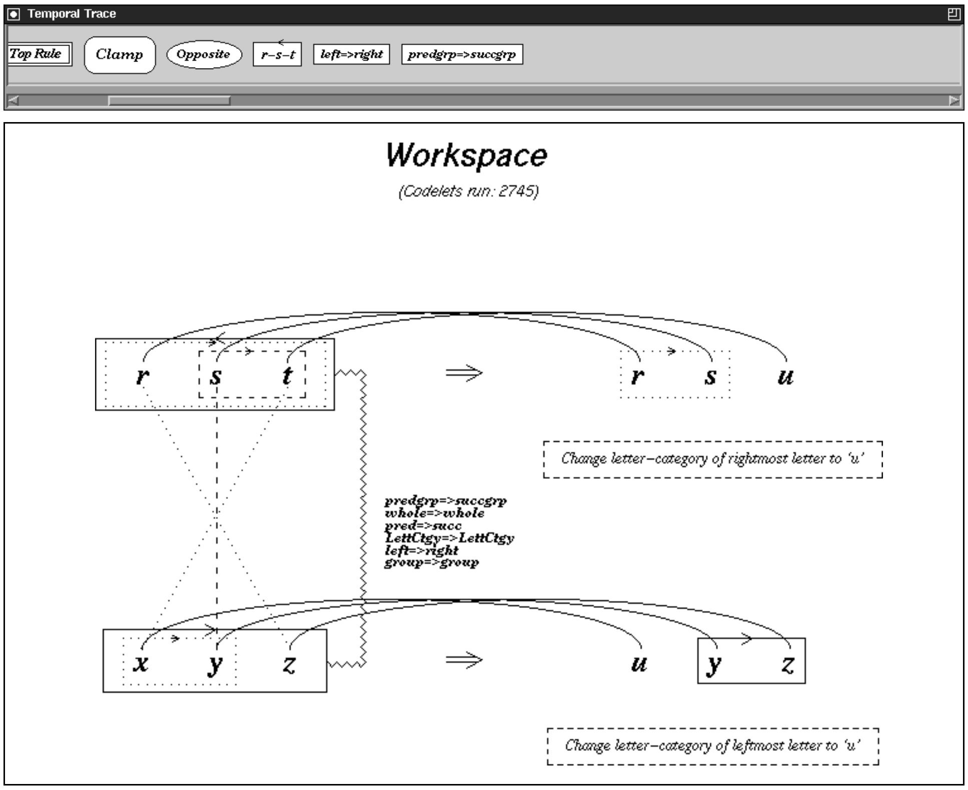
3-c


Temporal Trace
Identity
r–s–t
Top Rule
x–y–z
Bottom Rule
Bottom Rule
Clamp
Clamp
Top Rule
Event 9: Built rule
(Codelets run: 2060)
r s t ⇒ r s u
Change letter–category of rightmost letter to 'u'
succgrp=>succgrp
whole=>whole
succ=>succ
LettCtgy=>LettCtgy
right=>right
group=>group
x y z ⇒ u y z
1lmost=>lmost letter=>letter
2middle=>middle letter=>letter
3rmost=>rmost letter=>letter

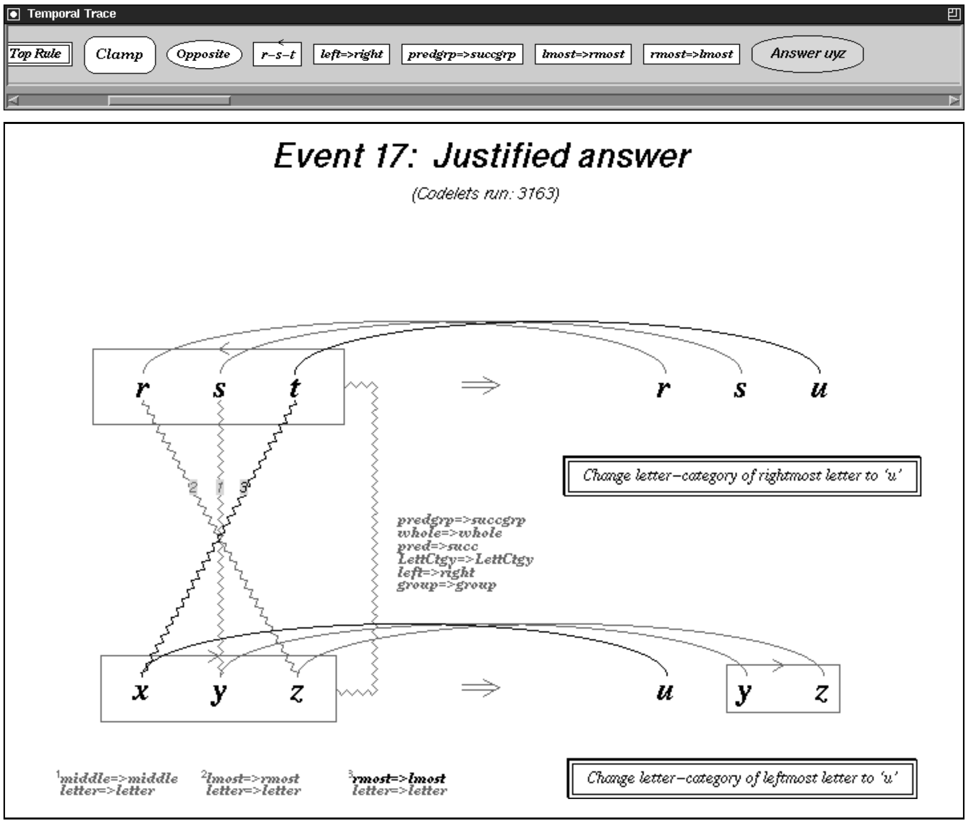
3-d


Temporal Trace
Identity
r–s–t
Top Rule
x–y–z
Bottom Rule
Bottom Rule
Clamp
Clamp
Top Rule
Clamp
Opposite
Event 10: Clamped patterns
(Codelets run: 2656)
r s t ⇒ r s u
Change letter–category of rightmost letter to 'u'
succgrp=>succgrp
whole=>whole
succ=>succ
LettCtgy=>LettCtgy
right=>right
group=>group
x y z ⇒ u y z
1lmost=>lmost letter=>letter
2middle=>middle letter=>letter
3rmost=>rmost letter=>letter
Change letter–category of leftmost letter to 'u'

3-e


Temporal Trace
Top Rule
Clamp
Opposite
r–s–t
left=>right
predgrp=>succgrp
Workspace
(Codelets run: 2745)
r s t
r s u
Change letter–category of rightmost letter to 'u'
predgrp=>succgrp
whole=>whole
pred=>succ
LettCtgy=>LettCtgy
left=>right
group=>group
x y z
u y z
Change letter–category of leftmost letter to 'u'

3-f


Temporal Trace
Top Rule
Clamp
Opposite
r–s–t
left=>right
predgrp=>succgrp
lmost=>rmost
rmost=>lmost
Answer uyz
Event 17: Justified answer
(Codelets run: 3163)
r s t
r s u
Change letter–category of rightmost letter to 'u'
predgrp=>succgrp
whole=>whole
pred=>succ
LettCtgy=>LettCtgy
left=>right
group=>group
x y z
u y z
1 middle=>middle letter=>letter
2 lmost=>rmost letter=>letter
3 rmost=>lmost letter=>letter
Change letter–category of leftmost letter to 'u'

5.2.2 Examples of jootsing

Run 4: *abc* ⇒ *abd; xyz* ⇒ *dyz*

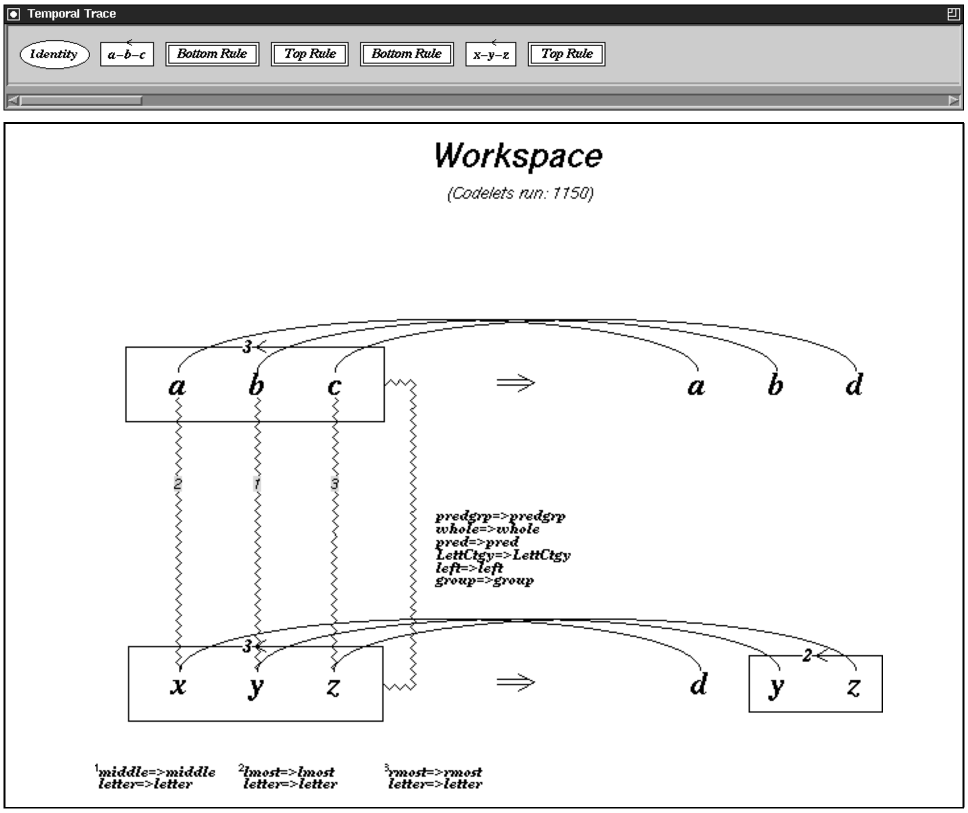
As Run 3 demonstrated, clamping the rule-codelet pattern shown in Panel *3-b* can facilitate the creation of new rules, but it is not guaranteed to do so every time. The next run is similar to the previous run, except that here the program is unable to justify the given answer, even after repeated clamping. Eventually, after three unsuccessful clamps, the program gives up.

In this run, the program attempts to justify the answer ***dyz*** to the problem "***abc*** ⇒ ***abd; xyz*** ⇒ ***?***". It begins, as before, by building a same-direction mapping between the initial string and target string. By time step 1150, the two top rules *Change letter-category of rightmost letter to successor* and *Change letter-category of letter 'c' to 'd'* have been created to describe ***abc*** ⇒ ***abd***, and the two bottom rules *Change letter-category of letter 'x' to 'd'* and *Change letter-category of leftmost letter to 'd'* have been created to describe ***xyz*** ⇒ ***dyz*** (see Panel *4-a*).

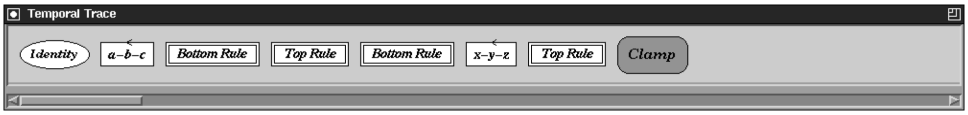
None of these rules, however, can be unified. Consequently, the program is unable to make any more progress. The situation remains essentially unchanged until time step 1882, when a *Progress-watcher* codelet notices the lack of Workspace activity and, in response, clamps the rule-codelet pattern shown earlier (see Panel *4-b*). Unfortunately, as before, no new rules are discovered, so the program tries again at time step 2354 (see Panel *4-c*). This second clamp again results in no progress, so a third clamp ensues at time step 2892 (see Panel *4-d*). This time, the new bottom rule *Change string to "dyz"* is discovered (see Panel *4-e*), but this does not help the situation very much. The top rule required for unification to occur remains elusive (*i.e.*, *Change letter-category of rightmost letter to 'd'*).

By time step 3228—approximately 300 codelets later—the program has still not

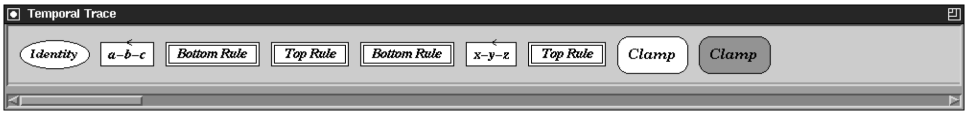
4-a


Temporal Trace
Identity
a–b–c
Bottom Rule
Top Rule
Bottom Rule
x–y–z
Top Rule
Workspace
(Codelets run: 1150)
a b c ⇒ a b d
x y z ⇒ d y z
predgrp=>predgrp
whole=>whole
pred=>pred
LettCtgy=>LettCtgy
left=>left
group=>group
1middle=>middle letter=>letter
2lmost=>lmost letter=>letter
3rmost=>rmost letter=>letter

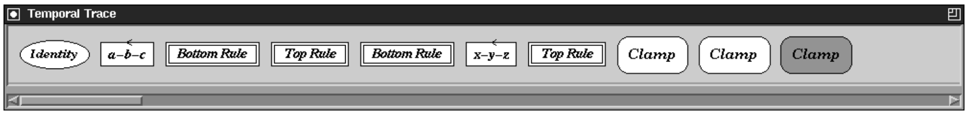
4-b


Temporal Trace
Identity
a–b–c
Bottom Rule
Top Rule
Bottom Rule
x–y–z
Top Rule
Clamp

4-c

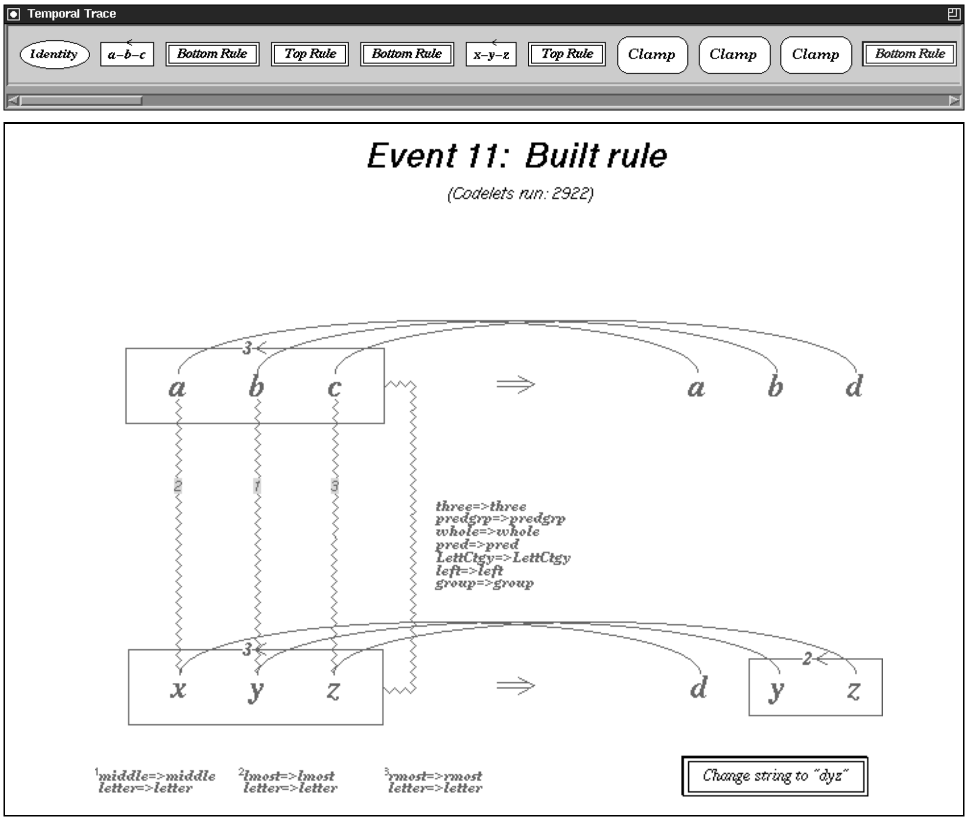

Temporal Trace
Identity
a–b–c
Bottom Rule
Top Rule
Bottom Rule
x–y–z
Top Rule
Clamp
Clamp

4-d


Temporal Trace
Identity
a–b–c
Bottom Rule
Top Rule
Bottom Rule
x–y–z
Top Rule
Clamp
Clamp
Clamp

discovered this rule. At this point, a *Jootser* codelet notices that the three rule-codelet clamp events in the Temporal Trace have all achieved little or no progress, so it decides to end the apparently futile cycle of clamping, at which point the program stops. Panel *4-f* shows the program's commentary from the time of the third clamp to the end of the run. As in the previous example, the clamps in this run actually arise from a lack of high-quality (*i.e.*, abstract) bottom rules, even though the real reason for the program's impasse is the lack of a top rule that can be unified with one of the existing bottom rules.

4-e



4-f

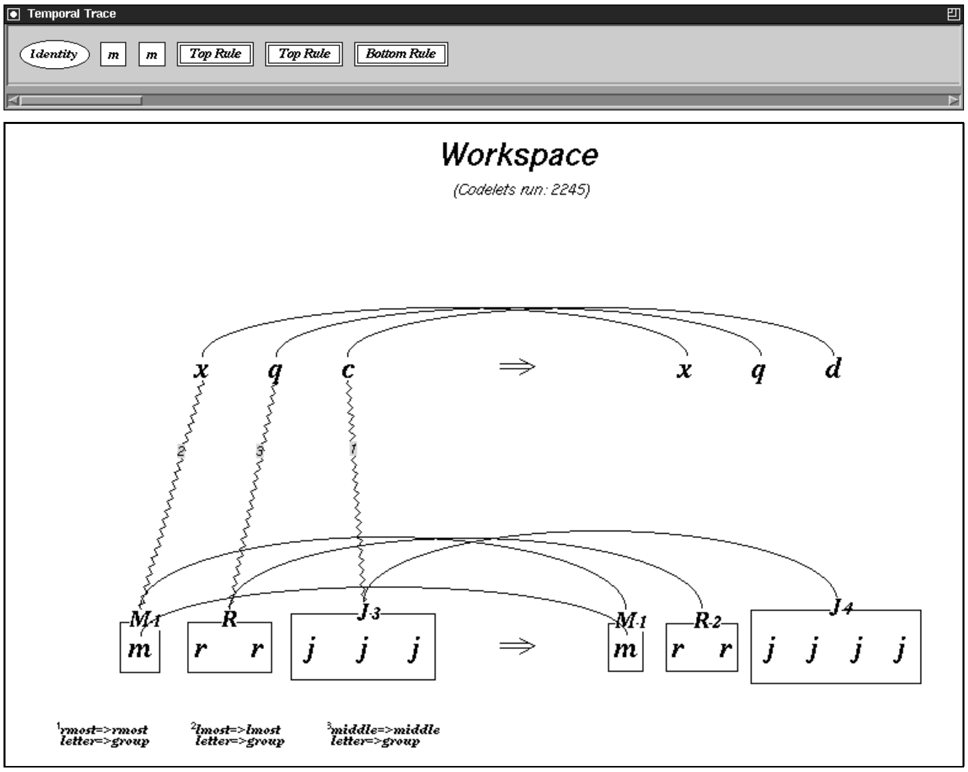
> *I'm getting frustrated. I still don't see a good way to describe how "xyz" changes to "dyz".*
>
> *I'll just have to try a little harder...*
>
> *Well, my latest effort to think up new rules resulted in very little progress. Guess it was not such a great effort, in retrospect.*
>
> *I just can't seem to come up with any better rules.*
>
> *Excuse me -- I think I'll go get some more punch.*

Run 5: ***xqc*** $\Rightarrow$ ***xqd; mrrjjj*** $\Rightarrow$ ***mrrjjjj***

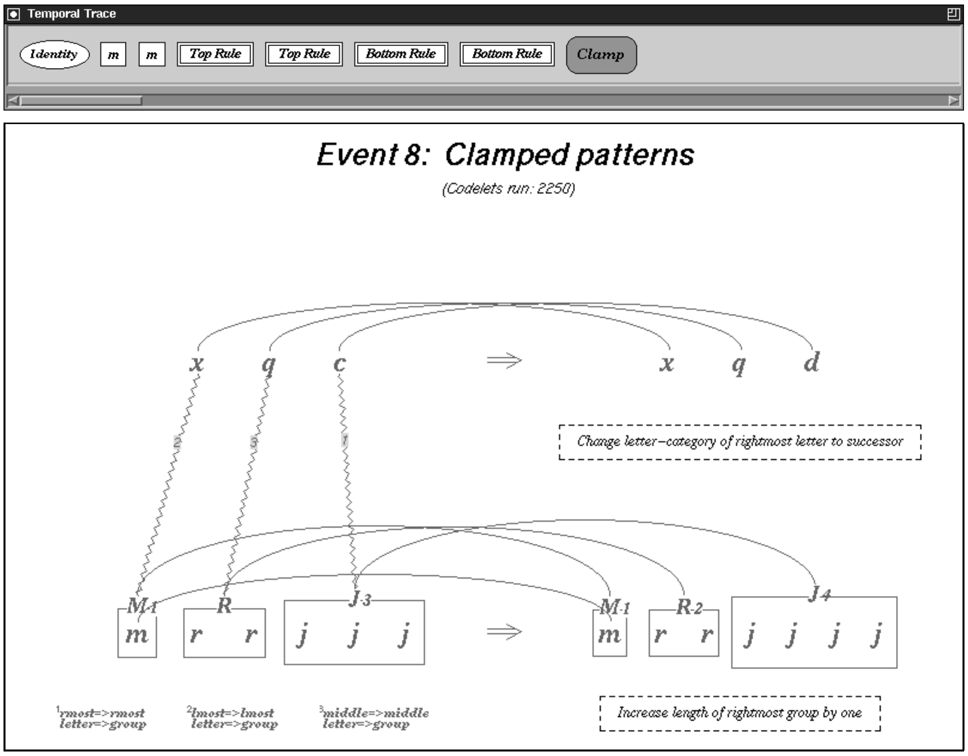
The next example illustrates jootsing from repeated justify clamping. In this run, the program is asked to justify the answer ***mrrjjjj*** to the problem *"**xqc** $\Rightarrow$ **xqd; mrrjjj** $\Rightarrow$?"*. At the outset, the run proceeds in much the same way as Run 1 discussed earlier. The sameness groups ***rr***, ***jjj***, and ***jjjj*** are quickly created in ***mrrjjj*** and ***mrrjjjj***, leading to the perception of ***m*** as a single-letter group in both strings. Strong vertical and horizontal same-direction mappings are created between all of the strings, giving rise to the top rules *Change letter-category of rightmost letter to successor* and *Change letter-category of letter 'c' to 'd'*, and the bottom rule *Change length of rightmost group to four.* Moreover, by time step 2245, *Length* descriptions have been attached to most of the groups in ***mrrjjj*** and ***mrrjjjj*** (see Panel *5-a*).

Soon afterwards, the bottom rule *Increase length of rightmost group by one* is built, which leads almost immediately to a justify clamp involving this rule and the first top rule (see Panel *5-b*). By the end of the clamp period at time step 2506, a *Length* description has been attached to the ***rr*** group in ***mrrjjj***, but the *1–2–3* structure of the string as a whole has not yet been noticed. The program thus tries again at time step 2637, clamping the same pair of rules as before in an effort to induce a *Letter-Category* $\Rightarrow$ *Length* slippage, which is needed in order to make the rules inter-translatable (see Panel *5-c*).

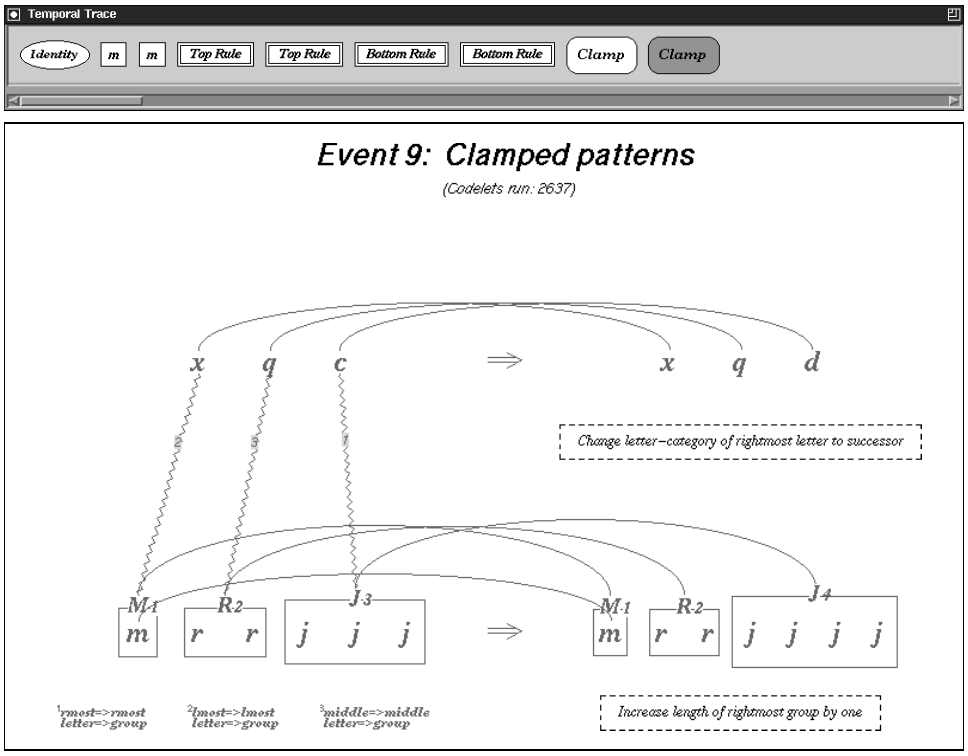
5-a


Temporal Trace
Identity
m
m
Top Rule
Top Rule
Bottom Rule
Workspace
(Codelets run: 2245)
x q c ⇒ x q d
M1 R J3 ⇒ M1 R2 J4
m r r j j j ⇒ m r r j j j j
¹rmost=>rmost letter=>group
²lmost=>lmost letter=>group
³middle=>middle letter=>group

5-b


Temporal Trace
Identity
m
m
Top Rule
Top Rule
Bottom Rule
Bottom Rule
Clamp
Event 8: Clamped patterns
(Codelets run: 2250)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M1 R J3 ⇒ M1 R2 J4
m r r j j j ⇒ m r r j j j j
¹rmost=>rmost letter=>group
²lmost=>lmost letter=>group
³middle=>middle letter=>group
Increase length of rightmost group by one

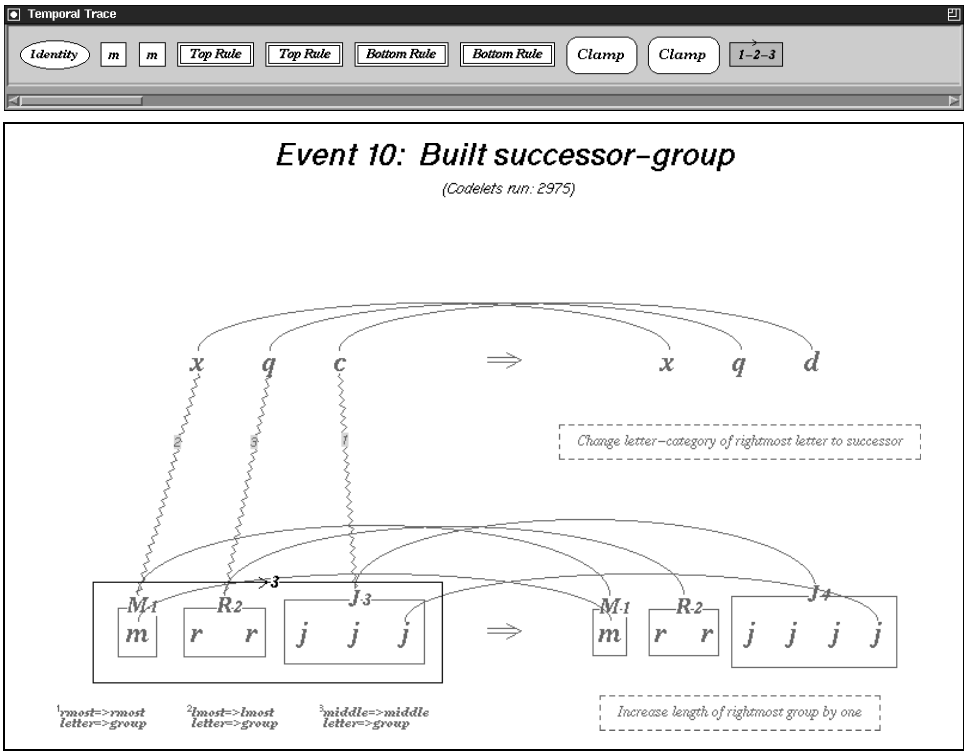
5-c



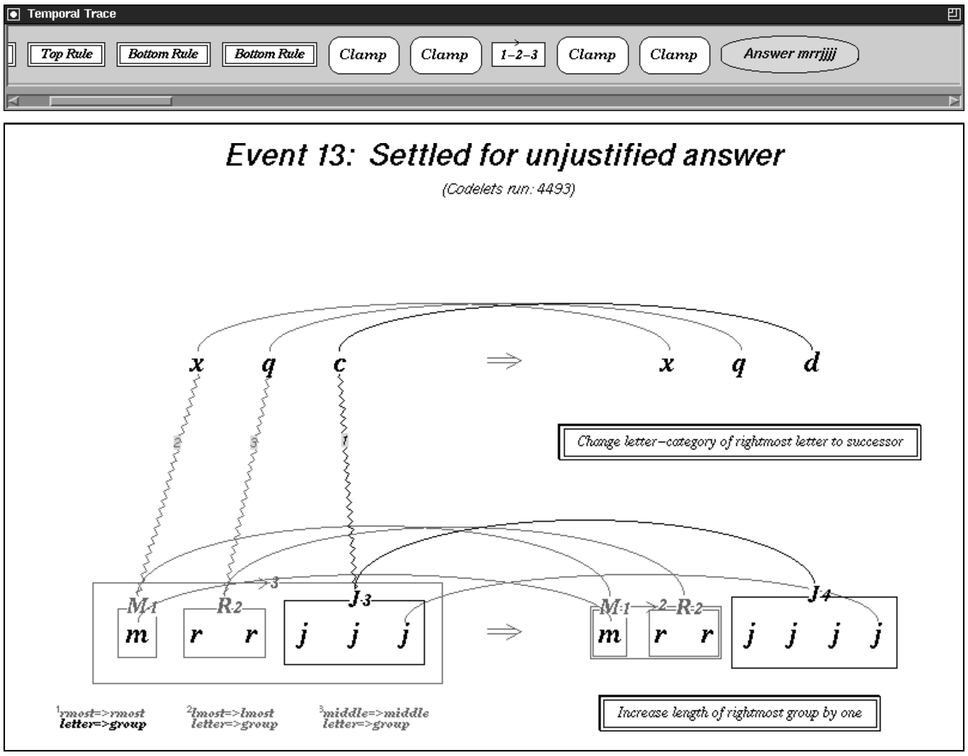
The program perceives ***mrrjjj*** as a successor group during this second clamp period, at time step 2975 (see Panel *5-d*). However, as was discussed in Chapter 4, no *Letter-Category* ⇒ *Length* slippage is possible between ***xqc*** and ***mrrjjj***, since ***xqc*** cannot be seen as a successor group under any circumstances. Nevertheless, the program keeps trying. It continues to clamp the same pair of rules until, after two more unsuccessful clamps, it notices its own pattern of repetitive behavior. Consequently, at time step 4493, it decides to give up on trying to make sense of the answer (see Panel *5-e*). At this point, it creates an *unjustified* answer description for ***mrrjjjj*** containing an unjustified *Bond-Facet: different* vertical theme (in addition to a justified *String-Position: identity* theme), which it then stores in memory in the usual way.[7]

[7]The unjustified vertical themes *Group-Type: identity* and *Direction: identity* are also included, since the *Bond-Facet: different* theme implies that ***xqc*** and ***mrrjjj*** are both groups. This facilitates the generation of English-language commentary when comparing the unjustified answer ***mrrjjjj*** to other answers in memory.

5-d


Temporal Trace
Identity
m
m
Top Rule
Top Rule
Bottom Rule
Bottom Rule
Clamp
Clamp
1–2–3
Event 10: Built successor-group
(Codelets run: 2975)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M1 R2 J3
m r r j j j ⇒ m r r j j j j
M1 R2 J4
Increase length of rightmost group by one
rmost=>rmost letter=>group
lmost=>lmost letter=>group
middle=>middle letter=>group

5-e


Temporal Trace
Top Rule
Bottom Rule
Bottom Rule
Clamp
Clamp
1–2–3
Clamp
Clamp
Answer mrrjjjj
Event 13: Settled for unjustified answer
(Codelets run: 4493)
x q c ⇒ x q d
Change letter–category of rightmost letter to successor
M1 R2 J3
m r r j j j ⇒ m r r j j j j
M1 R2 J4
Increase length of rightmost group by one
rmost=>rmost letter=>group
lmost=>lmost letter=>group
middle=>middle letter=>group

5-f

> *Looks like that last brilliant idea I had resulted in zero progress. Guess it was a pretty useless idea, in retrospect.*
>
> *Aha! I have another idea...*
>
> *Looks like that last brilliant idea I had resulted in zero progress. Guess it was a pretty useless idea, in retrospect.*
>
> *Okay, I'm stumped. This answer makes no sense to me. I see no way to make the necessary letter–category <=> length slippage here.*

Panel *5-f* shows the program's commentary regarding its final two clamp attempts (both of which achieved no progress) and its subsequent failure to justify ***mrrjjjj***.

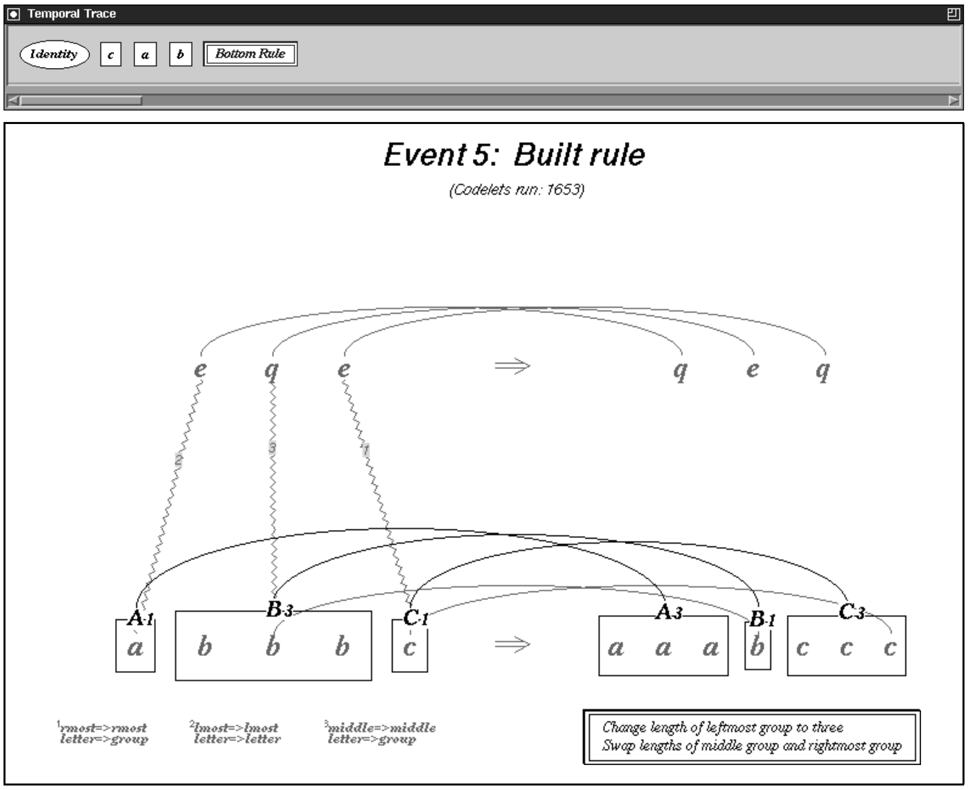
Run 6: *eqe* ⇒ *qeq; abbbc* ⇒ *aaabccc*

The next example is similar in flavor to Run 5. In this run, the program tries to justify the answer ***aaabccc*** to the problem *"**eqe** ⇒ **qeq; abbbc** ⇒ **?**"* in two different ways, but eventually gives up after trying unsuccessfully several times to make a *Letter-Category* ⇒ *Length* slippage between ***eqe*** and ***abbbc***.

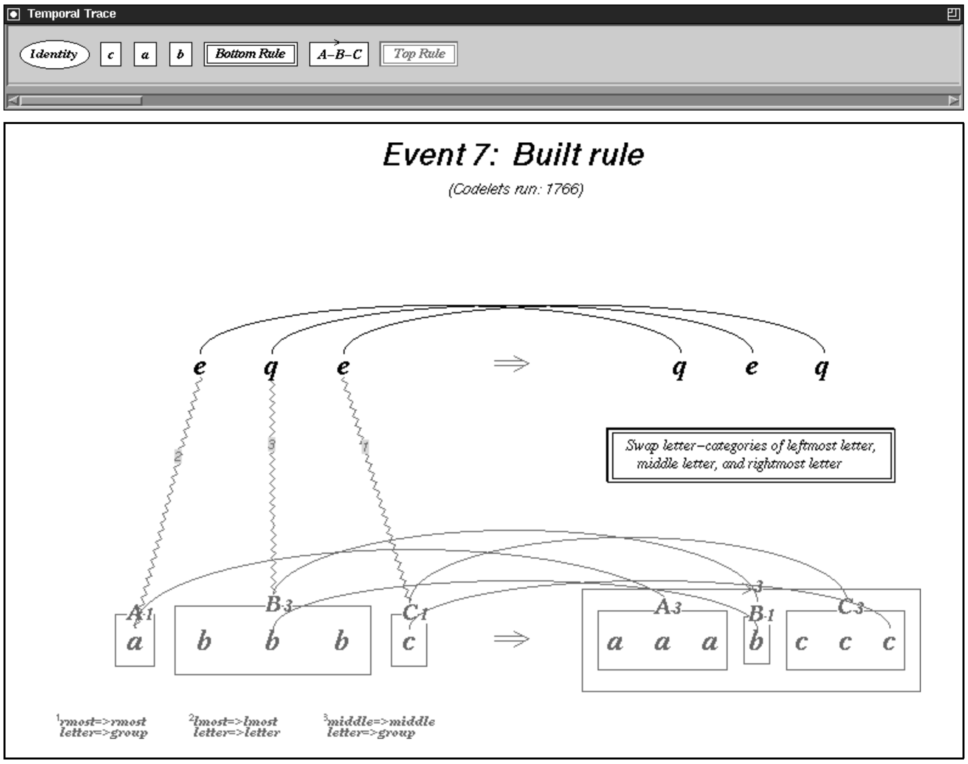
By time step 1650, the program has perceived the letters ***a*** and ***c*** in ***abbbc***, and the letter ***b*** in ***aaabccc***, as single-letter groups, and has attached *Length* descriptions to all of the groups in the strings, including the sameness groups ***bbb***, ***aaa***, and ***ccc***. The program's first attempt at describing the ***abbbc*** ⇒ ***aaabccc*** change, however, results in a somewhat odd bottom rule (see Panel *6-a*).

During the next 1100 time steps, ***abbbc*** and ***aaabccc*** are both perceived as successor groups (on the basis of letter-categories rather than group-lengths), and several other top and bottom rules are created (see Panels *6-b* through *6-e*). Eventually, at time step 2740, the bottom rule *Swap lengths of leftmost group, middle group, and rightmost group* is created, which can be unified with the top rule shown in Panel *6-b*. Accordingly, at time step 2874, the program clamps a set of patterns based on these rules in an attempt to induce a *Letter-Category* ⇒ *Length* slippage between ***eqe*** and

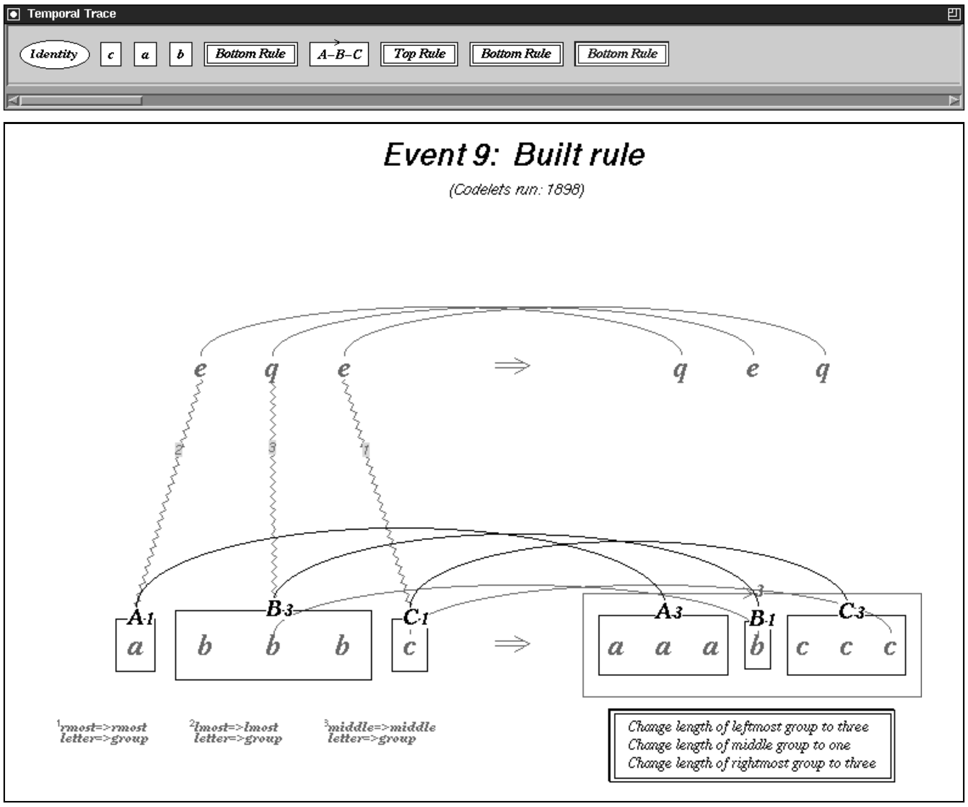
6-a


Temporal Trace
Identity
c
a
b
Bottom Rule
Event 5: Built rule
(Codelets run: 1653)
e q e ⇒ q e q
A1
B3
C1
a b b b c ⇒ a a a b c c c
A3
B1
C3
[1]rmost=>rmost letter=>group
[2]lmost=>lmost letter=>letter
[3]middle=>middle letter=>group
Change length of leftmost group to three
Swap lengths of middle group and rightmost group

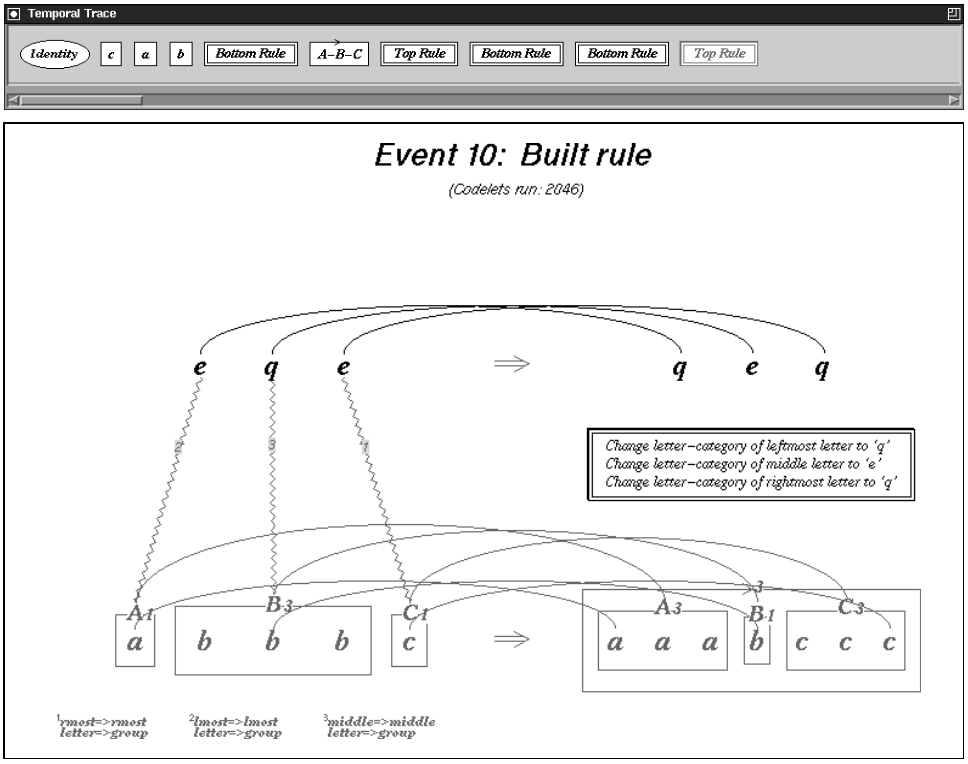
6-b


Temporal Trace
Identity
c
a
b
Bottom Rule
A–B–C
Top Rule
Event 7: Built rule
(Codelets run: 1766)
e q e ⇒ q e q
Swap letter–categories of leftmost letter, middle letter, and rightmost letter
A1
B3
C1
a b b b c ⇒ a a a b c c c
A3
B1
C3
[1]rmost=>rmost letter=>group
[2]lmost=>lmost letter=>letter
[3]middle=>middle letter=>group

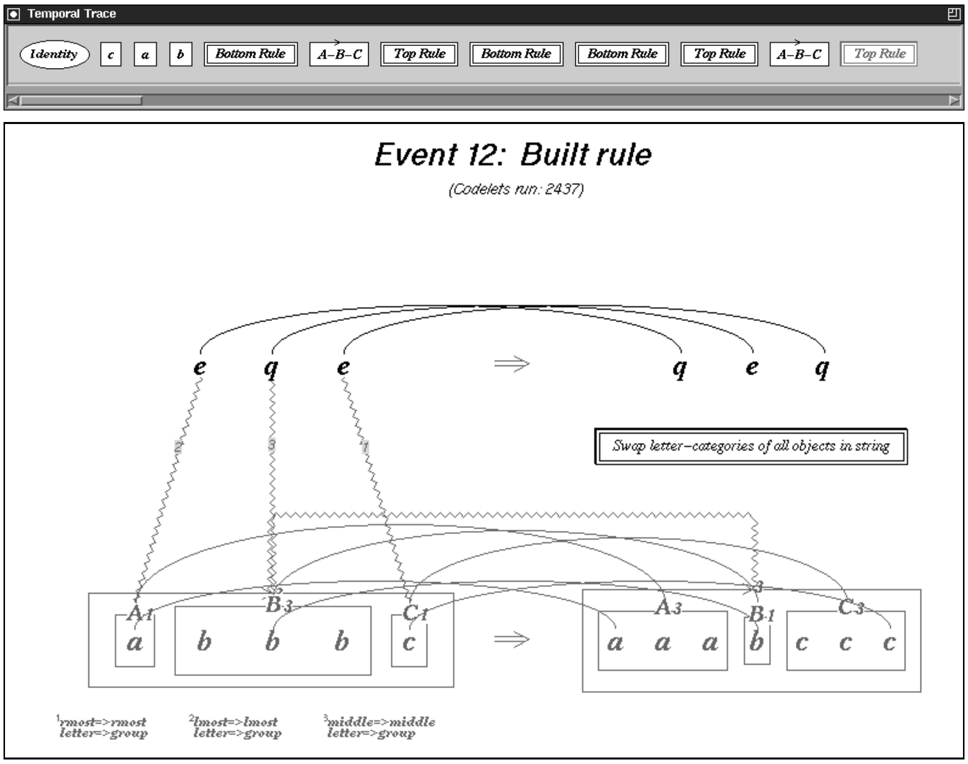
6-c


Temporal Trace
Identity
c
a
b
Bottom Rule
A–B–C
Top Rule
Bottom Rule
Bottom Rule
Event 9: Built rule
(Codelets run: 1898)
e q e ⇒ q e q
A1 B3 C1 ⇒ A3 B1 C3
a b b b c ⇒ a a a b c c c
1rmost=>rmost letter=>group
2lmost=>lmost letter=>group
3middle=>middle letter=>group
Change length of leftmost group to three
Change length of middle group to one
Change length of rightmost group to three

6-d


Temporal Trace
Identity
c
a
b
Bottom Rule
A–B–C
Top Rule
Bottom Rule
Bottom Rule
Top Rule
Event 10: Built rule
(Codelets run: 2046)
e q e ⇒ q e q
Change letter–category of leftmost letter to 'q'
Change letter–category of middle letter to 'e'
Change letter–category of rightmost letter to 'q'
A1 B3 C1 ⇒ A3 B1 C3
a b b b c ⇒ a a a b c c c
1rmost=>rmost letter=>group
2lmost=>lmost letter=>group
3middle=>middle letter=>group

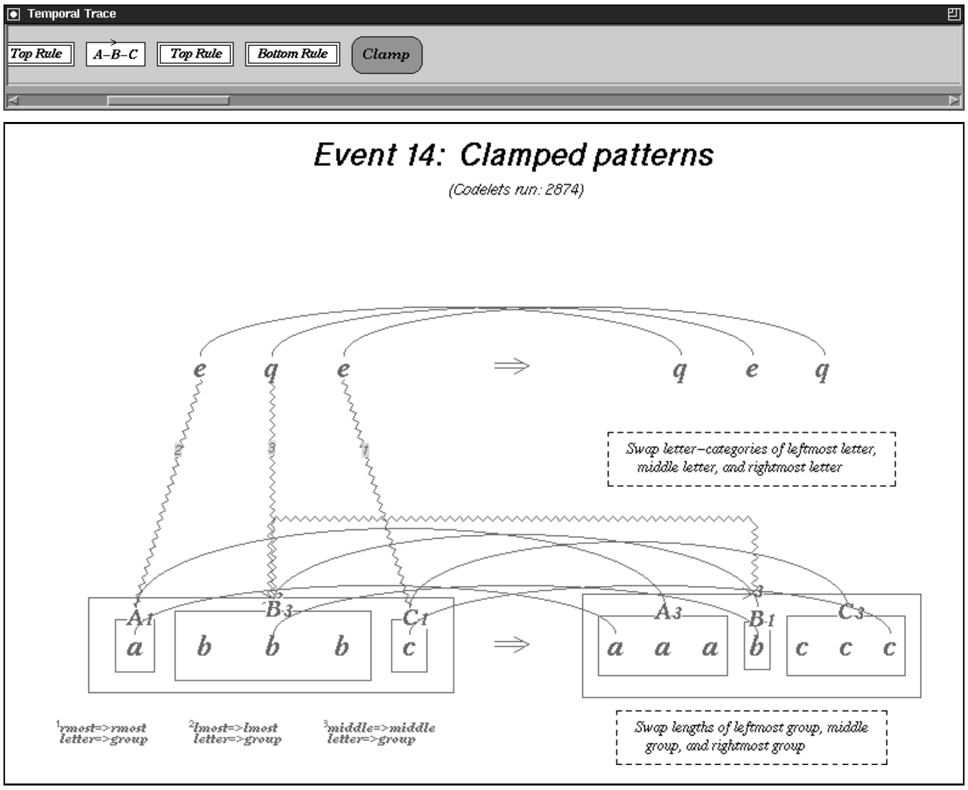
6-e



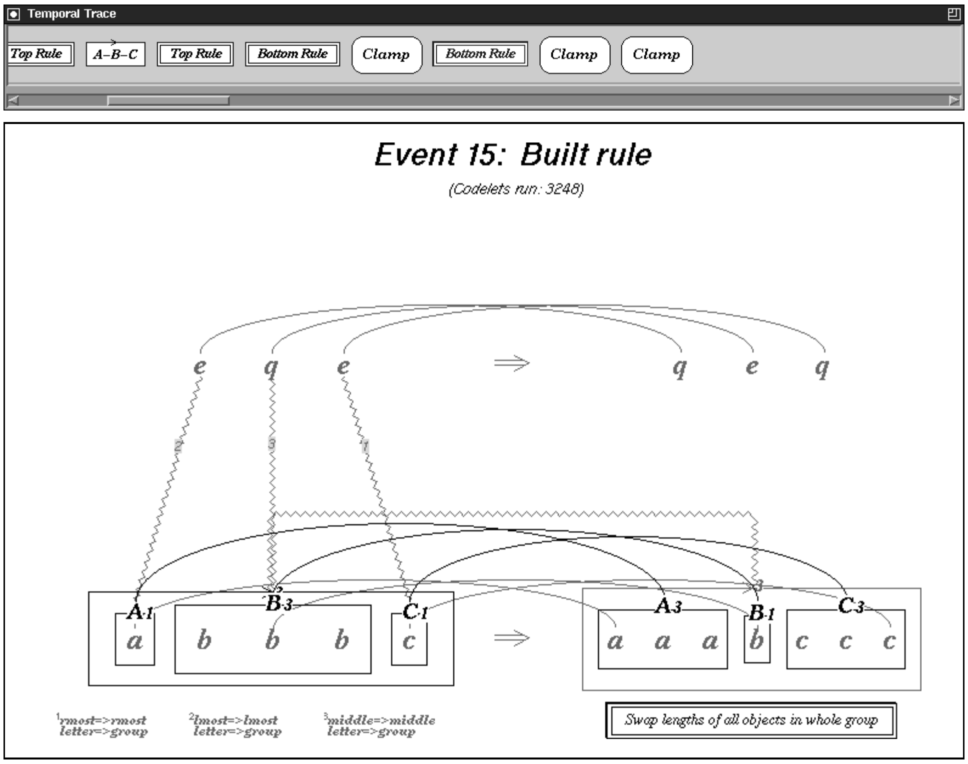
abbbc (see Panel *6-f*). This attempt is unsuccessful, so the program tries again at time steps 3379 and 3889. In the meantime, however, another rule for describing ***abbbc*** ⇒ ***aaabccc*** gets created in between clamp periods (see Panel *6-g*). This rule, which can be unified with the top rule shown in Panel *6-e*, offers the program a potential alternative route to justifying the answer ***aaabccc***. Therefore, it tries a few more clamps based on this pair of rules (see Panel *6-h*). Unfortunately, however, this effort fails too, leading the program to finally give up at time step 6196 (see Panel *6-i*).

Consequently, the answer description created for ***aaabccc*** includes the same set of unjustified vertical themes as the answer description for ***mrrjjjj*** in Run 5 (*i.e.*, *Bond-Facet: different*, *Group-Type: identity*, and *Direction: identity*), as well as the same justified *String-Position: identity* theme. These two answers are thus quite similar at an abstract level of description.

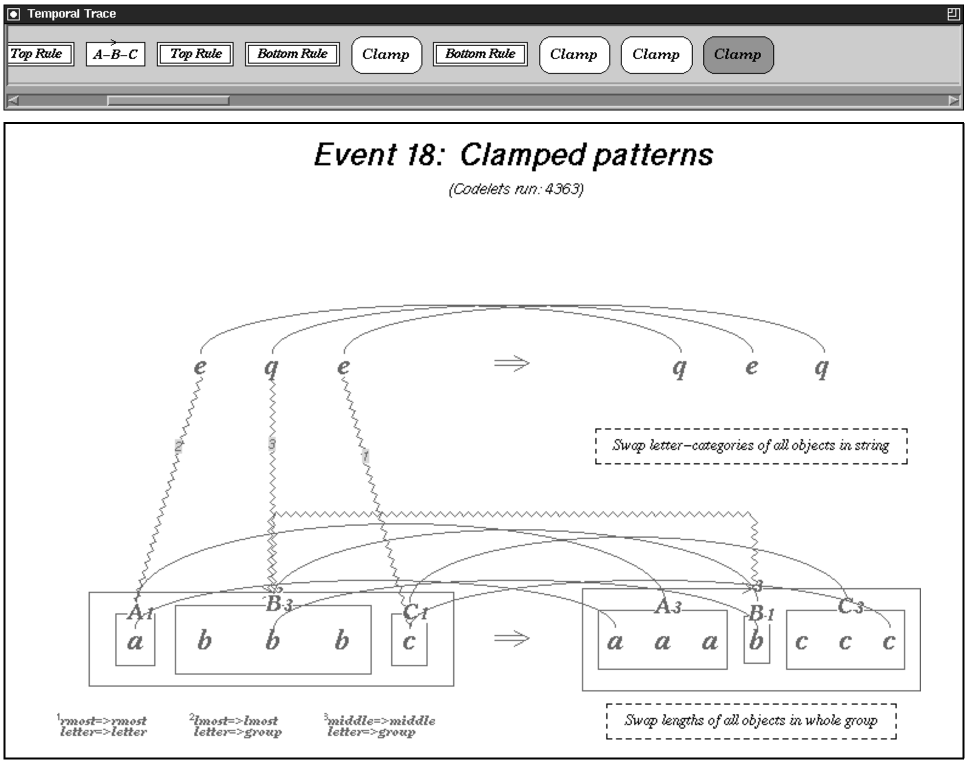
6-f


Temporal Trace
Top Rule
A–B–C
Top Rule
Bottom Rule
Clamp
Event 14: Clamped patterns
(Codelets run: 2874)
e q e ⇒ q e q
Swap letter–categories of leftmost letter, middle letter, and rightmost letter
A1 B3 C1
a b b b c ⇒ a a a b c c c
A3 B1 C3
rmost=>rmost letter=>group
lmost=>lmost letter=>group
middle=>middle letter=>group
Swap lengths of leftmost group, middle group, and rightmost group

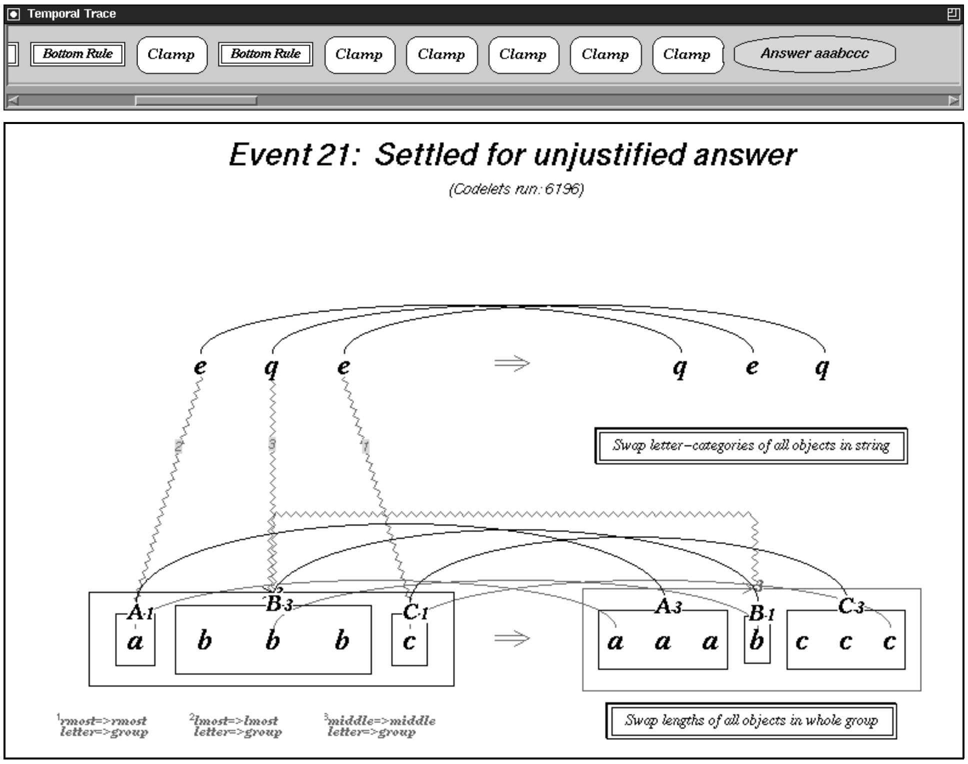
6-g


Temporal Trace
Top Rule
A–B–C
Top Rule
Bottom Rule
Clamp
Bottom Rule
Clamp
Clamp
Event 15: Built rule
(Codelets run: 3248)
e q e ⇒ q e q
A1 B3 C1
a b b b c ⇒ a a a b c c c
A3 B1 C3
rmost=>rmost letter=>group
lmost=>lmost letter=>group
middle=>middle letter=>group
Swap lengths of all objects in whole group

6-h


Temporal Trace
Top Rule
A–B–C
Top Rule
Bottom Rule
Clamp
Bottom Rule
Clamp
Clamp
Clamp
Event 18: Clamped patterns
(Codelets run: 4363)
e q e ⇒ q e q
Swap letter–categories of all objects in string
A1 B3 C1 A3 B1 C3
a b b b c ⇒ a a a b c c c
[1]rmost=>rmost letter=>letter
[2]lmost=>lmost letter=>group
[3]middle=>middle letter=>group
Swap lengths of all objects in whole group

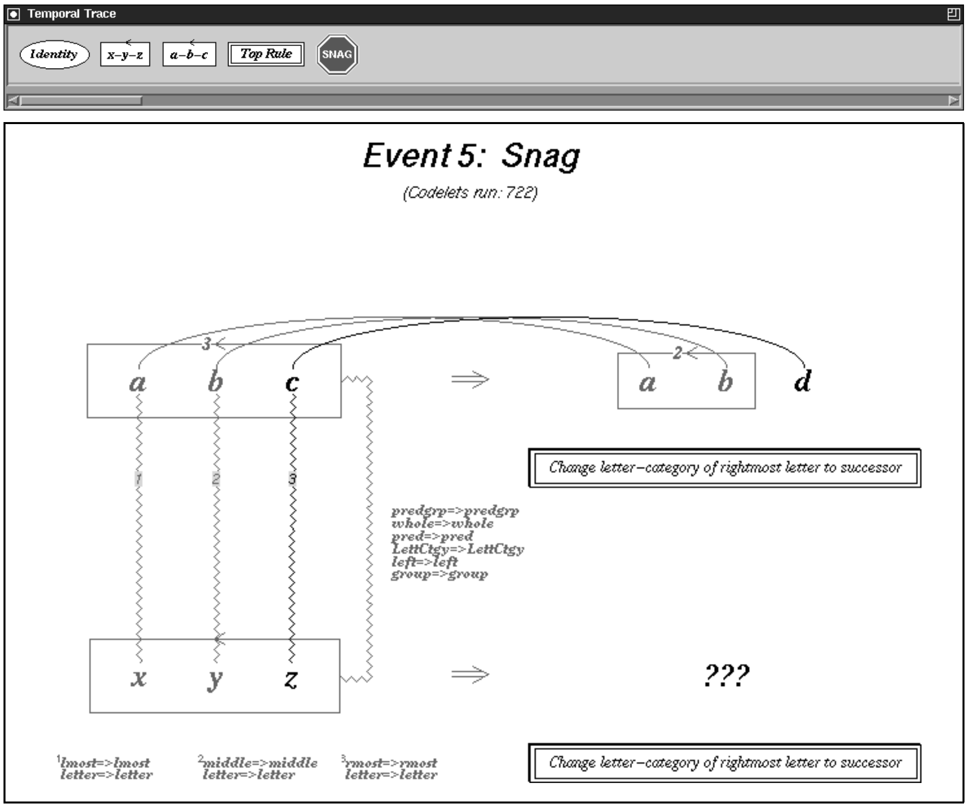
6-i


Temporal Trace
Bottom Rule
Clamp
Bottom Rule
Clamp
Clamp
Clamp
Clamp
Clamp
Answer aaabccc
Event 21: Settled for unjustified answer
(Codelets run: 6196)
e q e ⇒ q e q
Swap letter–categories of all objects in string
A1 B3 C1 A3 B1 C3
a b b b c ⇒ a a a b c c c
[1]rmost=>rmost letter=>group
[2]lmost=>lmost letter=>group
[3]middle=>middle letter=>group
Swap lengths of all objects in whole group

Run 7: ***abc*** ⇒ ***abd; xyz*** ⇒ ***?***

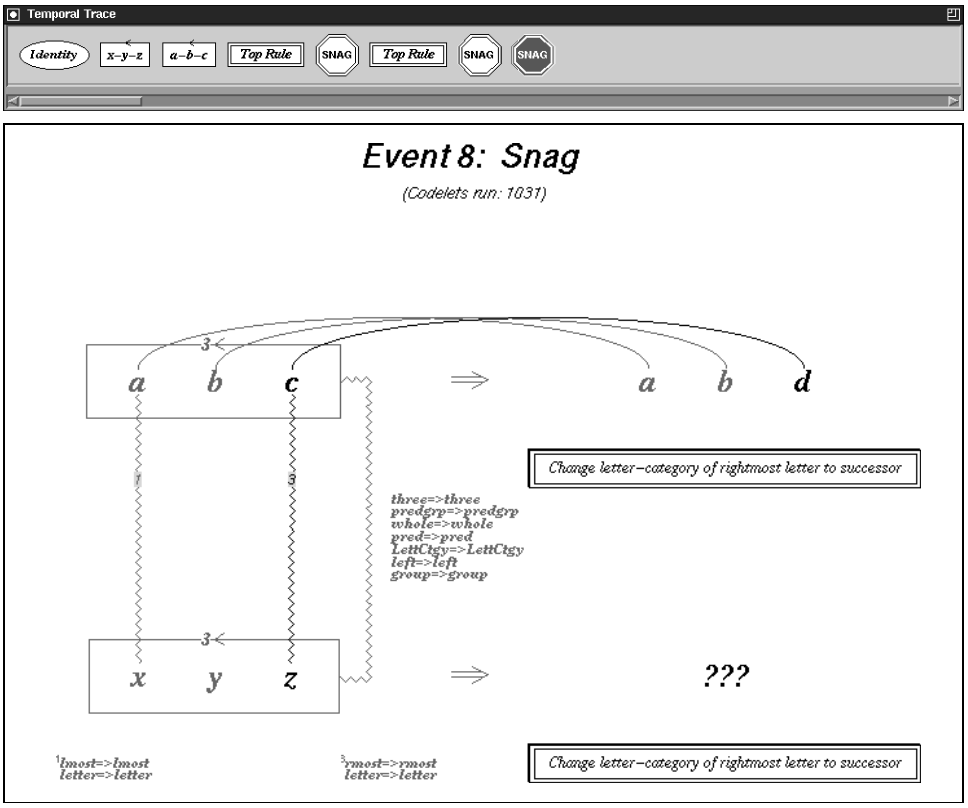
The next run demonstrates Metacat's ability to recognize when it is hitting the same snag over and over again, and to then try something different instead. In this run, the program is asked to come up with an answer on its own to the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*. Initially, the program perceives both ***abc*** and ***xyz*** as predecessor groups, and views the ***abc*** ⇒ ***abd*** change abstractly, according to the rule *Change letter-category of rightmost letter to successor*, which quickly leads to the usual snag involving the letter ***z*** (see Panel *7-a*).

7-a



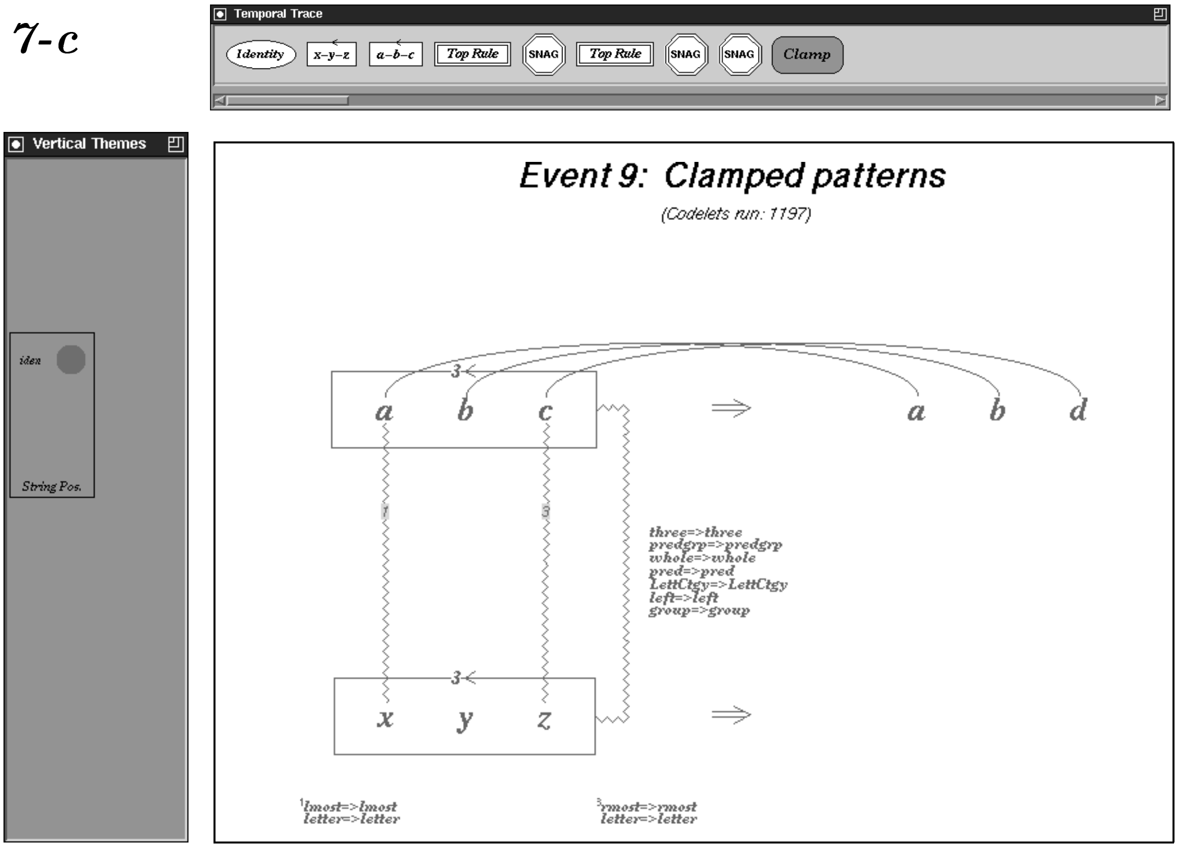
In the aftermath of the snag, the temperature shoots up to 100, various Workspace structures get broken, and the new rule *Change letter-category of rightmost letter to 'd'* is created. However, the program soon rebuilds the same structures as before, clinging to the view that the letter ***z*** in ***xyz*** corresponds to the letter ***c*** in ***abc***, and ignores the newly-created rule. Consequently, it hits the snag again at time step 904—and then again at time step 1031 (see Panel *7-b*).

7-b



In general, associated with each snag event in the Temporal Trace is a vertical theme-pattern that characterizes the way in which the target-string objects directly responsible for the snag are seen as corresponding to their counterpart objects in the initial string. In the present example, this pattern consists of the themes *String-Position: identity* and *Object-Type: identity*, based on the concept-mappings *rightmost* ⇒ *rightmost* and *letter* ⇒ *letter* underlying the ***c***–***z*** bridge. Thus all three snag events in the Trace involve exactly the same set of vertical themes—as well as the same rule—which eventually draws the attention of a *Jootser* codelet at time step 1197. In response to these identical snag events, the codelet probabilistically decides to negatively clamp the recurring *String-Position: identity* theme, in an effort to elicit

7-c



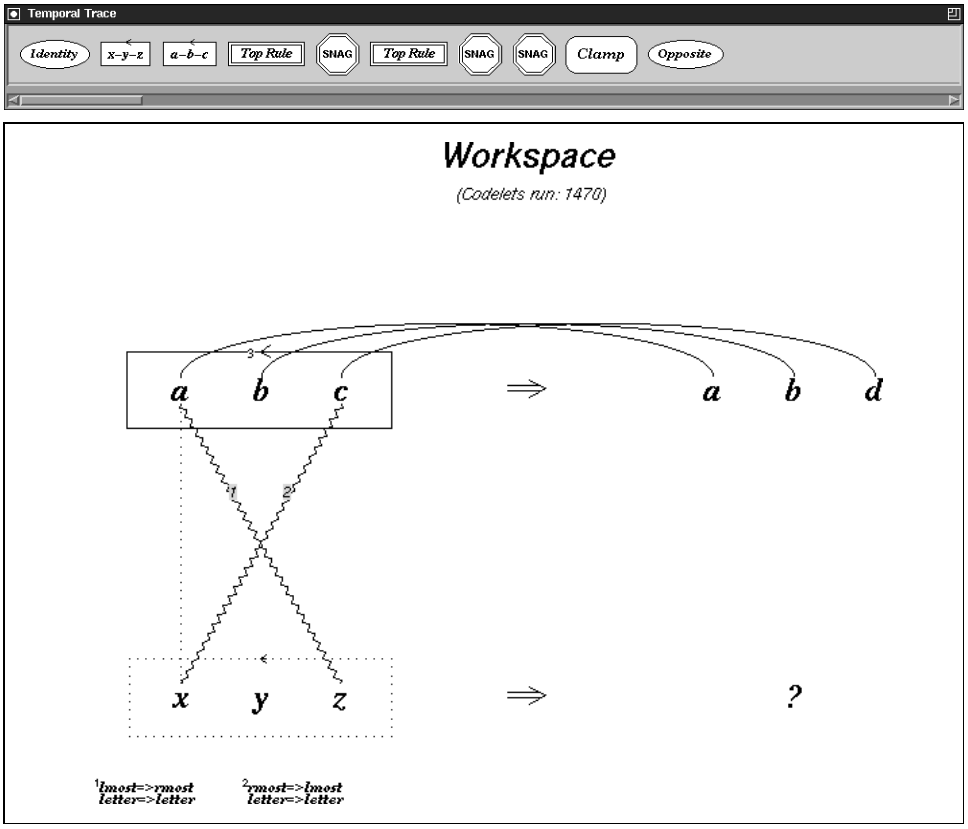
a reinterpretation of the problem that does not involve this idea (see Panel *7-c*).[8]

As a consequence, the ***a–x*** and ***c–z*** bridges are soon replaced by the new bridges ***a–z*** and ***c–x***, causing the concept of *opposite* in the Slipnet to become highly activated (see Panel *7-d*). Thus a new interpretation of ***abc*** and ***xyz*** begins to crystallize around the idea of oppositeness. However, even though ***a*** and ***z*** are now seen as corresponding to each other, their alphabetic symmetry remains unnoticed by the program, because *Alphabetic-Position* descriptions have not yet been attached to both of these letters.

As it turns out, the clamp period ends at time 1718 without much further progress having been achieved. In fact, the ***c–x*** bridge is broken shortly thereafter, threatening

[8]Negatively-clamped themes appear in red on the screen, although this is hard to see from the figure.

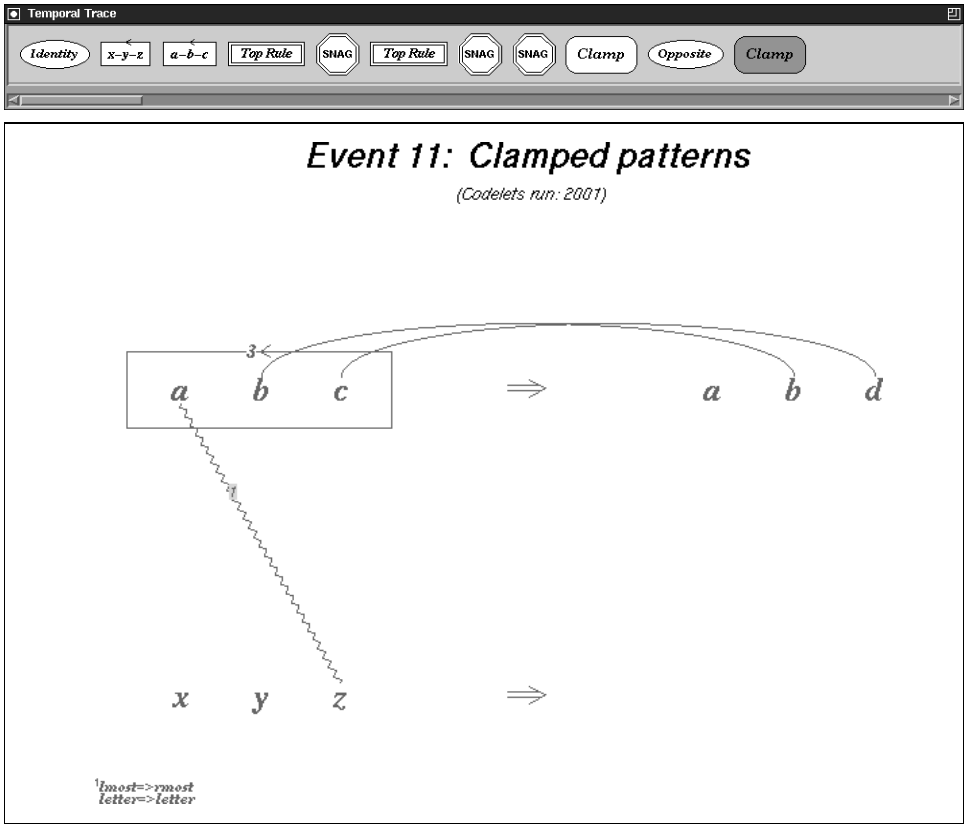
7-d



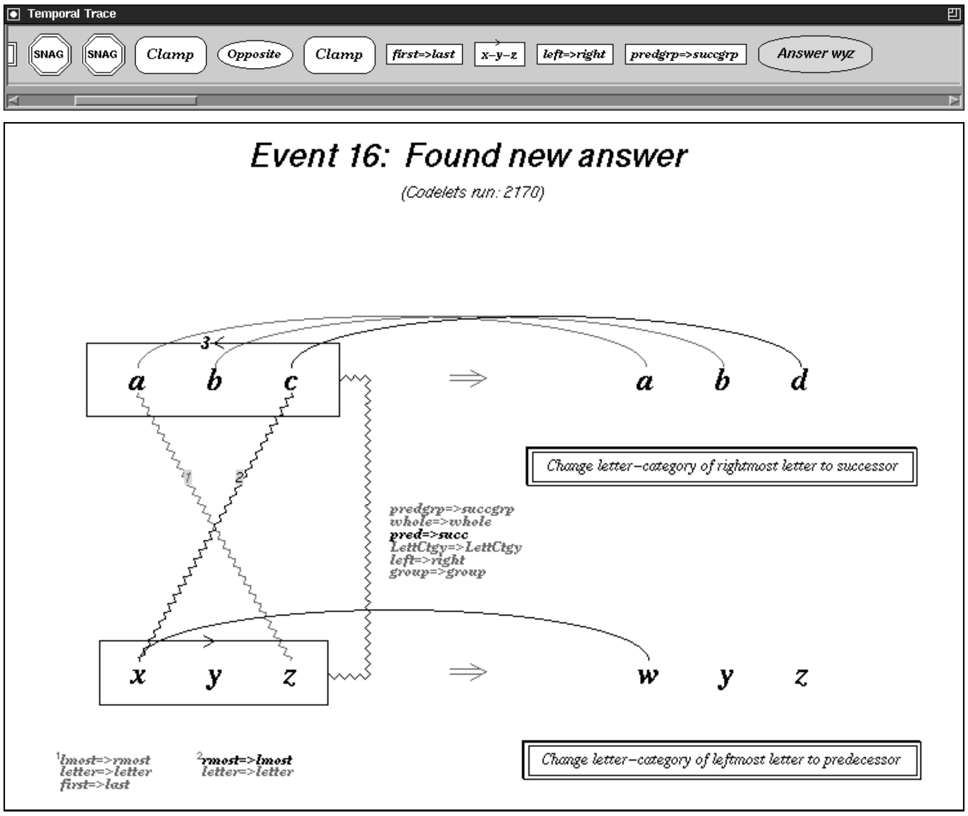
to undo the progress made so far. Fortunately, however, at time step 2001, another *Jootser* codelet clamps the same negative theme-pattern as before, giving the program another push in the right direction—that is, away from the ideas associated with the snag (see Panel *7-e*).

This time, the program notices the alphabetic symmetry between ***a*** and ***z***, because by now *Alphabetic-Position* descriptions exist for both letters (the most recent such description having been attached to the letter ***a*** at time step 1982). Accordingly, the program adds a *first* ⇒ *last* slippage to the ***a***–***z*** bridge at time step 2031. Soon afterwards, it perceives ***xyz*** as a successor group, paving the way for the creation of a fully symmetric mapping between ***abc*** and ***xyz*** based on the slippages *left* ⇒ *right* and *predecessor-group* ⇒ *successor-group*, which leads in turn to the discovery of the answer ***wyz*** at time step 2170 (see Panel *7-f*).

7-e


Temporal Trace
Identity
x-y-z
a-b-c
Top Rule
SNAG
Top Rule
SNAG
SNAG
Clamp
Opposite
Clamp
Event 11: Clamped patterns
(Codelets run: 2001)
3
a b c ⇒ a b d
1
x y z ⇒
[1]lmost=>rmost
letter=>letter

7-f


Temporal Trace
SNAG
SNAG
Clamp
Opposite
Clamp
first=>last
x-y-z
left=>right
predgrp=>succgrp
Answer wyz
Event 16: Found new answer
(Codelets run: 2170)
3
a b c ⇒ a b d
Change letter−category of rightmost letter to successor
1
2
predgrp=>succgrp
whole=>whole
pred=>succ
LettCtgy=>LettCtgy
left=>right
group=>group
x y z ⇒ w y z
Change letter−category of leftmost letter to predecessor
[1]lmost=>rmost
letter=>letter
first=>last
[2]rmost=>lmost
letter=>letter

Run 8: *eqe* ⇒ *qeq; abbbc* ⇒ *?*

The next example demonstrates the idea of "meta-jootsing". In this run, Metacat is asked to come up with an answer on its own to the problem *"**eqe** ⇒ **qeq**; **abbbc** ⇒ **?**"*. The program begins by perceiving the string ***abbbc*** as a successor group composed of the letter ***a***, the group ***bbb***, and the letter ***c***. It also creates the two top rules shown below to describe the ***eqe*** ⇒ ***qeq*** change:

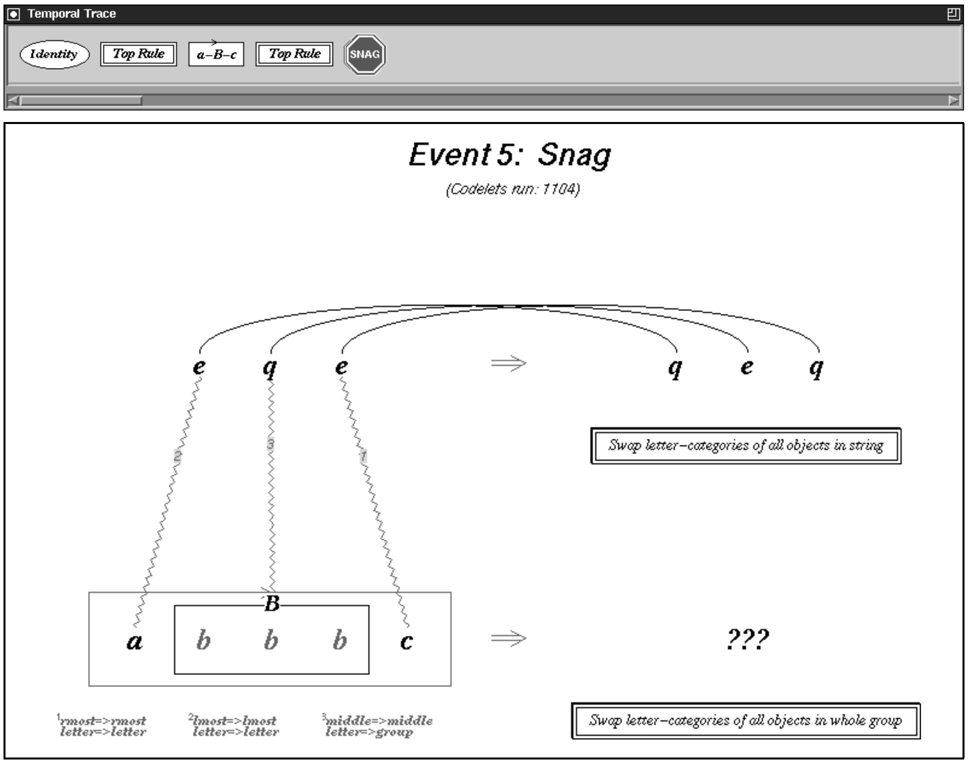
- *Swap letter-categories of all objects in string*
- *Change letter-category of leftmost letter to 'q'*
Change letter-category of middle letter to 'e'
Change letter-category of rightmost letter to 'q'

At time step 1104, it attempts to apply the first rule to ***abbbc***, which results in a snag, since the idea of "swapping" the letter-categories of ***a***, ***bbb***, and ***c*** makes no sense (see Panel *8-a*). Of course, if it had chosen to use the second rule instead of the first, then it would have found the answer ***qeeeq***, but it strongly prefers the first rule, since this rule is considerably more abstract (and succinct).

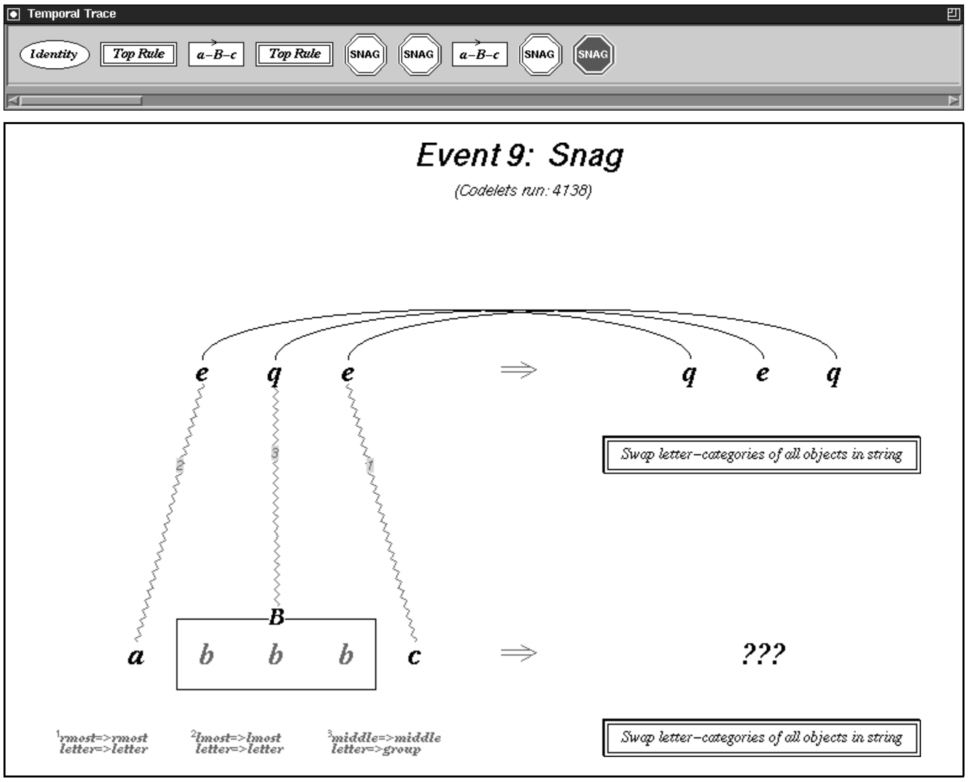
Over the next 3000 time steps, the program tries again and again to swap the letter-categories of ***abbbc***, often breaking various structures in the process, but always rebuilding them in the same way as before (see Panel *8-b*). Eventually, at time step 4280, a *Jootser* codelet notices the pattern of repeated snag events in the Temporal Trace, all of which involve the vertical themes *String-Position: identity*, *Object-Type: identity*, and *Object-Type: different*. These themes arise from the concept-mappings underlying the vertical bridges associated with the three snag objects ***a***, ***bbb***, and ***c*** (*i.e.*, *leftmost* ⇒ *leftmost*, *middle* ⇒ *middle*, *rightmost* ⇒ *rightmost*, *letter* ⇒ *letter* and *letter* ⇒ *group*). As it happens, the codelet probabilistically decides to negatively clamp just the *Object-Type: identity* theme (see Panel *8-c*).

In the aftermath of the clamp, ***abbbc*** is reperceived as a predecessor group, and a new top rule, *Swap letter-categories of leftmost letter, middle letter, and rightmost*

8-a


Temporal Trace
Identity
Top Rule
a–B–c
Top Rule
SNAG
Event 5: Snag
(Codelets run: 1104)
e q e ⇒ q e q
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

8-b


Temporal Trace
Identity
Top Rule
a–B–c
Top Rule
SNAG
SNAG
a–B–c
SNAG
SNAG
Event 9: Snag
(Codelets run: 4138)
e q e ⇒ q e q
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 string

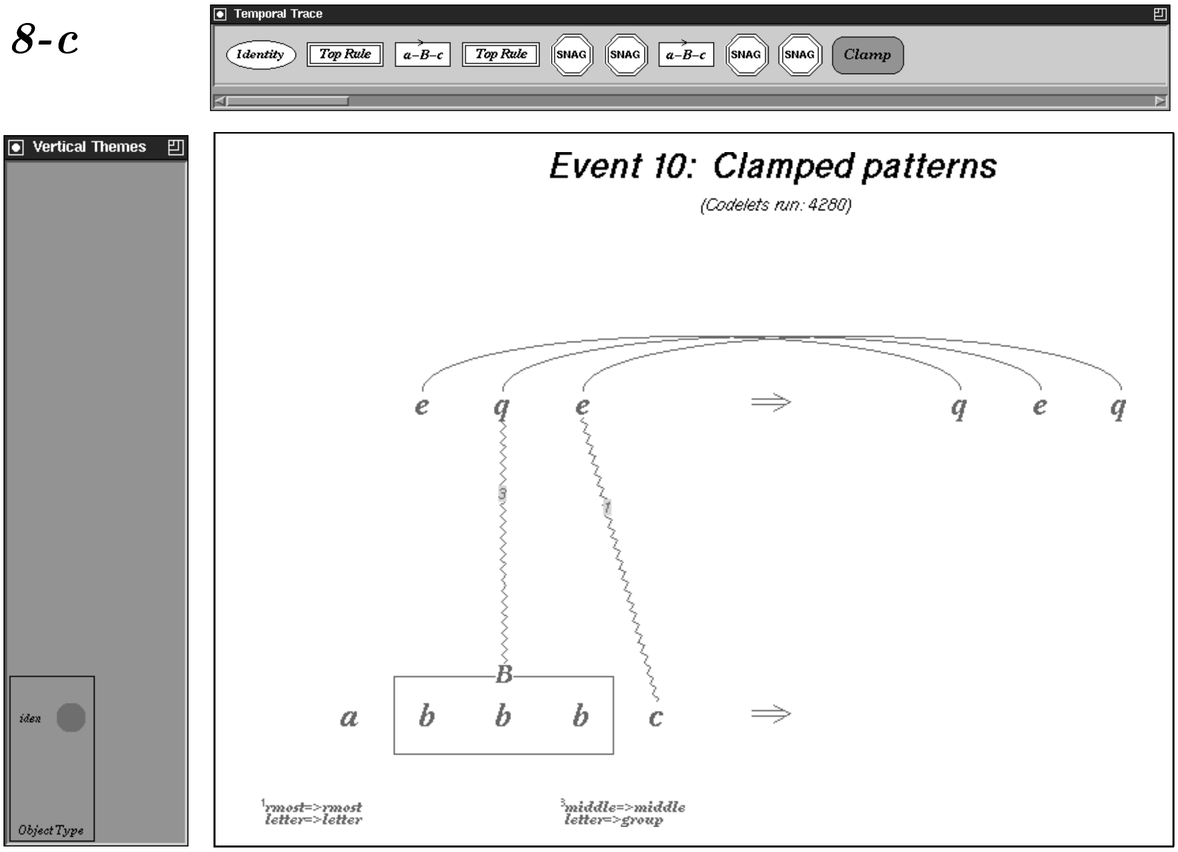
letter, is created. However, these new structures do not really change the basic situation. In any case, at time step 5040, another *Jootser* codelet decides to negatively clamp *both Object-Type* themes (see Panel *8-d*), which essentially "paralyzes" the program for the duration of the clamp, since any new vertical bridges created would be incompatible with one of the clamped themes.[9]

At time step 5487, the program hits the snag again. This is followed shortly thereafter by another snag-response clamp, this time involving the negative themes *String-Position: identity* and *Object-Type: identity* (see Panel *8-e*).

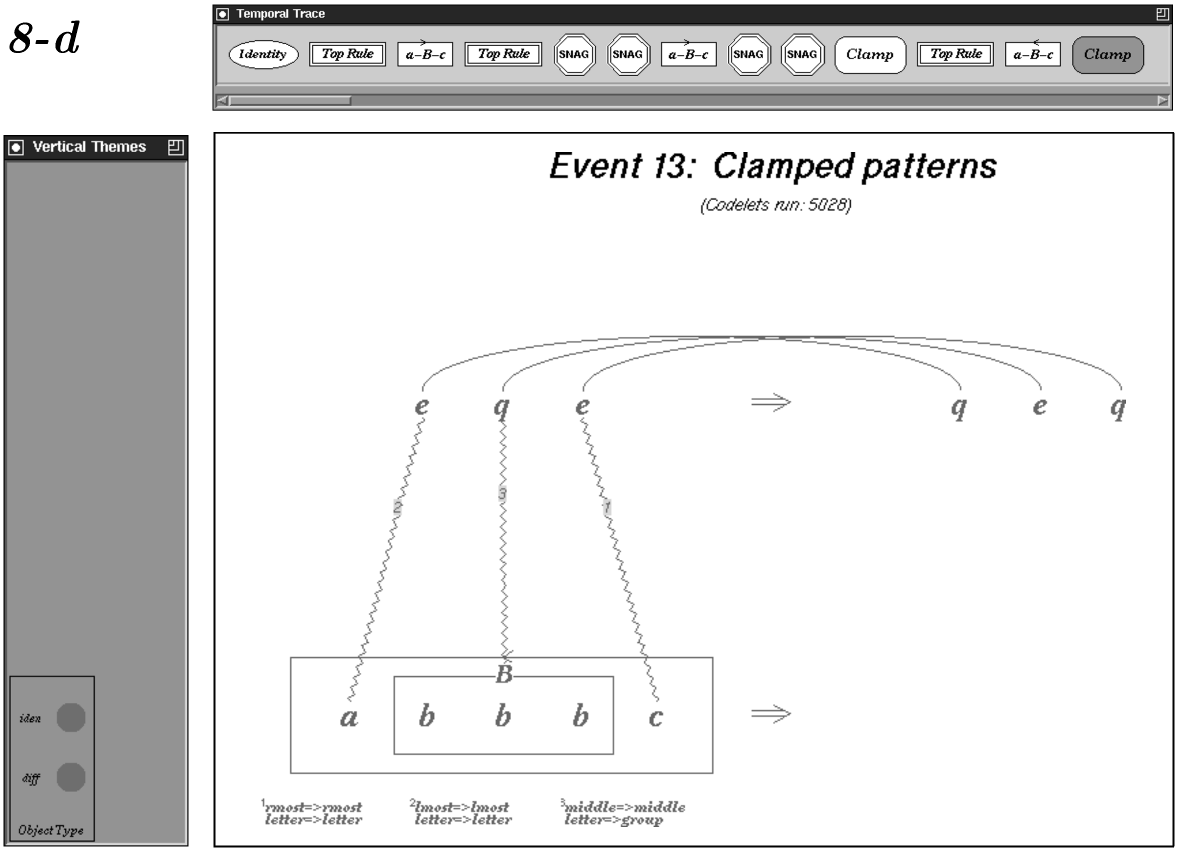
This clamp, like the one before it, achieves no new progress, since no new structures are created in its wake. Therefore, after hitting the snag yet again at time step 5904, the program finally decides to give up. More precisely, at time step 5933, a *Jootser* codelet notices the three clamp events in the Trace, all of which have theme-patterns that overlap to some degree. Furthermore, neither of the two most recent clamp events have resulted in any discernible progress. Consequently, the codelet prints a final parting message and then ends the run. Panel *8-f* shows the program's final commentary regarding its last unsuccessful clamp attempt.

[9]Negatively clamping more than one theme of the same category may sometimes be useful, however, if several different theme relations are possible for the category.

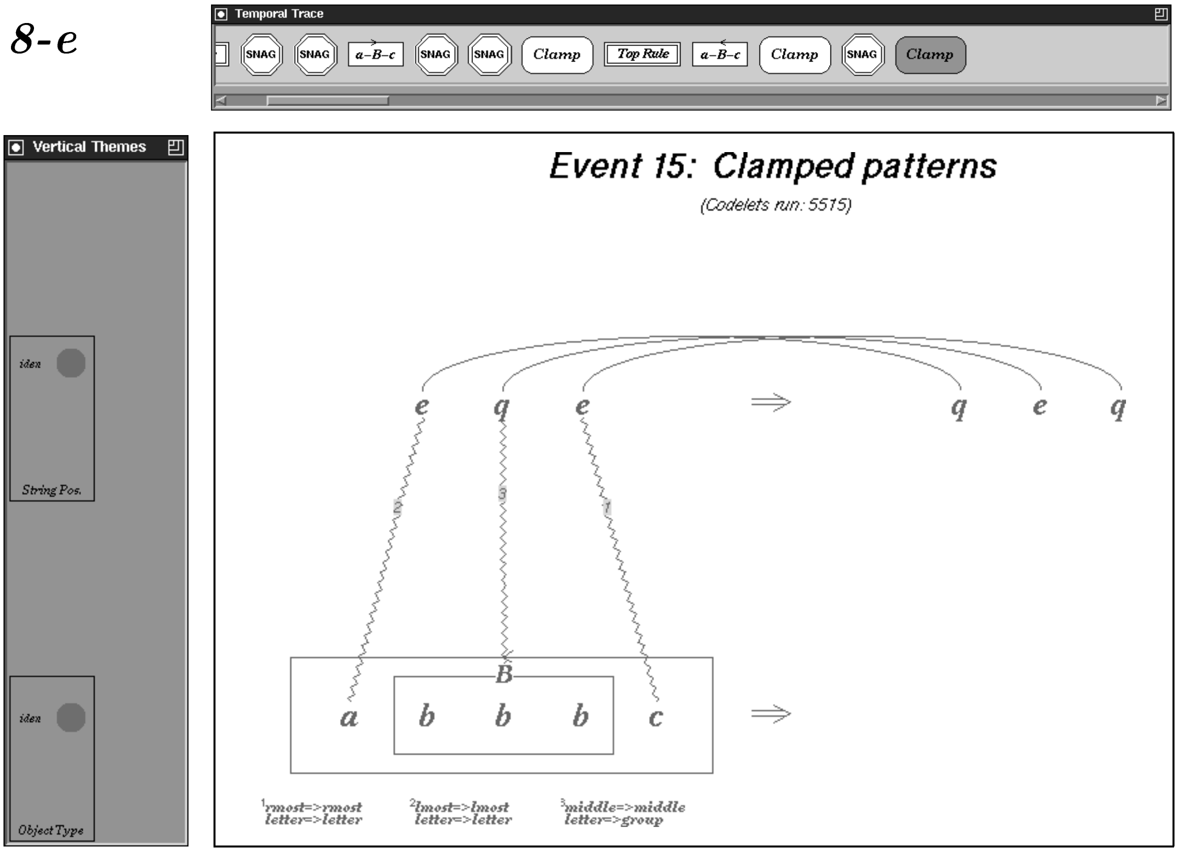
8-c


Temporal Trace
Identity
Top Rule
a–B–c
Top Rule
SNAG
SNAG
a–B–c
SNAG
SNAG
Clamp
Vertical Themes
iden
Object Type
Event 10: Clamped patterns
(Codelets run: 4280)
e q e ⇒ q e q
a b b b c ⇒
B
[1]rmost=>rmost
letter=>letter
[3]middle=>middle
letter=>group

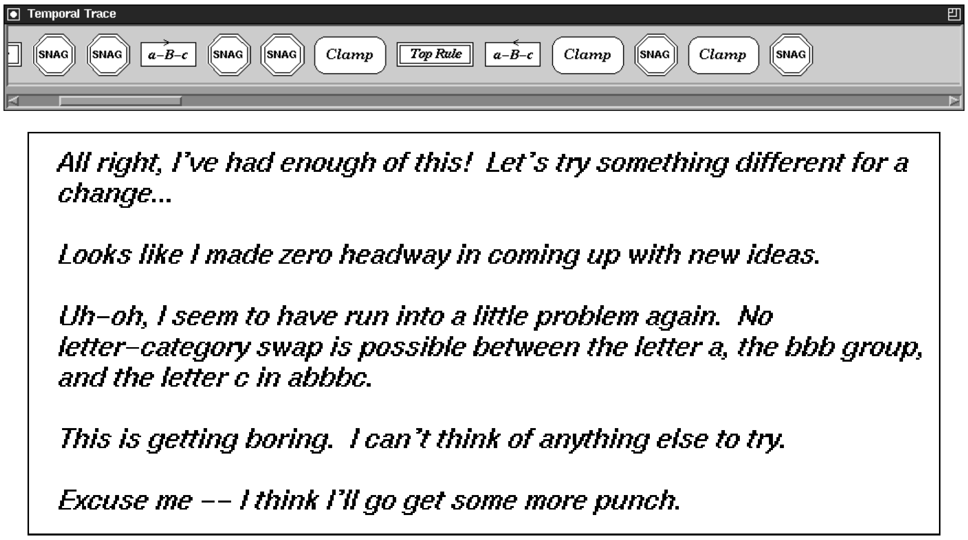
8-d


Temporal Trace
Identity
Top Rule
a–B–c
Top Rule
SNAG
SNAG
a–B–c
SNAG
SNAG
Clamp
Top Rule
a–B–c
Clamp
Vertical Themes
iden
diff
Object Type
Event 13: Clamped patterns
(Codelets run: 5028)
e q e ⇒ q e q
a b b b c ⇒
B
[1]rmost=>rmost
letter=>letter
[2]lmost=>lmost
letter=>letter
[3]middle=>middle
letter=>group

8-e


Temporal Trace
SNAG
SNAG
a–B–c
SNAG
SNAG
Clamp
Top Rule
a–B–c
Clamp
SNAG
Clamp
Vertical Themes
iden
String Pos.
iden
Object Type
Event 15: Clamped patterns
(Codelets run: 5515)
e q e ⇒ q e q
B
a b b b c ⇒
1 rmost=>rmost letter=>letter
2 lmost=>lmost letter=>letter
3 middle=>middle letter=>group

8-f


Temporal Trace
SNAG
SNAG
a–B–c
SNAG
SNAG
Clamp
Top Rule
a–B–c
Clamp
SNAG
Clamp
SNAG
All right, I've had enough of this! Let's try something different for a change...
Looks like I made zero headway in coming up with new ideas.
Uh-oh, I seem to have run into a little problem again. No letter-category swap is possible between the letter a, the bbb group, and the letter c in abbbc.
This is getting boring. I can't think of anything else to try.
Excuse me -- I think I'll go get some more punch.

5.2.3 Examples of answer comparison and reminding

This section presents a sampler of Metacat's explanations of the similarities and differences between several of the analogies from section 5.1. To generate these explanations, the program was first run (in justify mode) on each of the problems and answers shown in Figure 5.3. (Some of these runs were discussed in detail in the previous section.) In the figure, the lines connecting various pairs of answers indicate which answers reminded the program of other answers during this process. A faint dotted line indicates that Metacat was "vaguely" reminded of one answer by another, a darker dashed line indicates that it was "somewhat" reminded of an answer, and a heavy solid line indicates that it was "strongly" reminded of an answer.[10] For example, the answer ***wyz*** to the problem *"**rst**⇒ **rsu**; **xyz**⇒ **?**"* reminded the program "vaguely" of the answer ***uyz*** to the same problem, and "somewhat" of the answer ***wyz*** to the problem *"**abc**⇒ **abd**; **xyz**⇒ **?**"* (as was shown earlier in Figure 4.17 of Chapter 4).

In the case of the answer ***aaabccc*** to the problem *"**eqe**⇒ **qeq**; **abbbc**⇒ **?**"*, however, the degree of activation of other answers depends on whether or not Metacat has tried this problem on its own before, and therefore knows that it leads to a snag (as was discussed in sections 4.7.2 and 4.7.3 of Chapter 4). The program was thus run twice on this answer. The first time around, it was given ***aaabccc*** to justify before it had ever tried to swap the letter-categories of ***abbbc*** on its own. In this case, the program was strongly reminded of the answer ***aaabaaa*** to the problem *"**eqe**⇒ **qeq**; **abbba**⇒ **?**"*, and somewhat reminded of the answer ***mrrjjjj*** to the problem *"**xqc**⇒ **xqd**; **mrrjjj**⇒ **?**"* (both of which it had seen before). These remindings are marked with an asterisk (*) in Figure 5.3.

Next, the just-found answer ***aaabccc*** was manually deleted from memory, and

[10]These terms, which reflect the activation levels of answer descriptions, correspond respectively to the numerical ranges 1–30, 31–70, and 71–100. No reminding occurs in the case of zero activation.

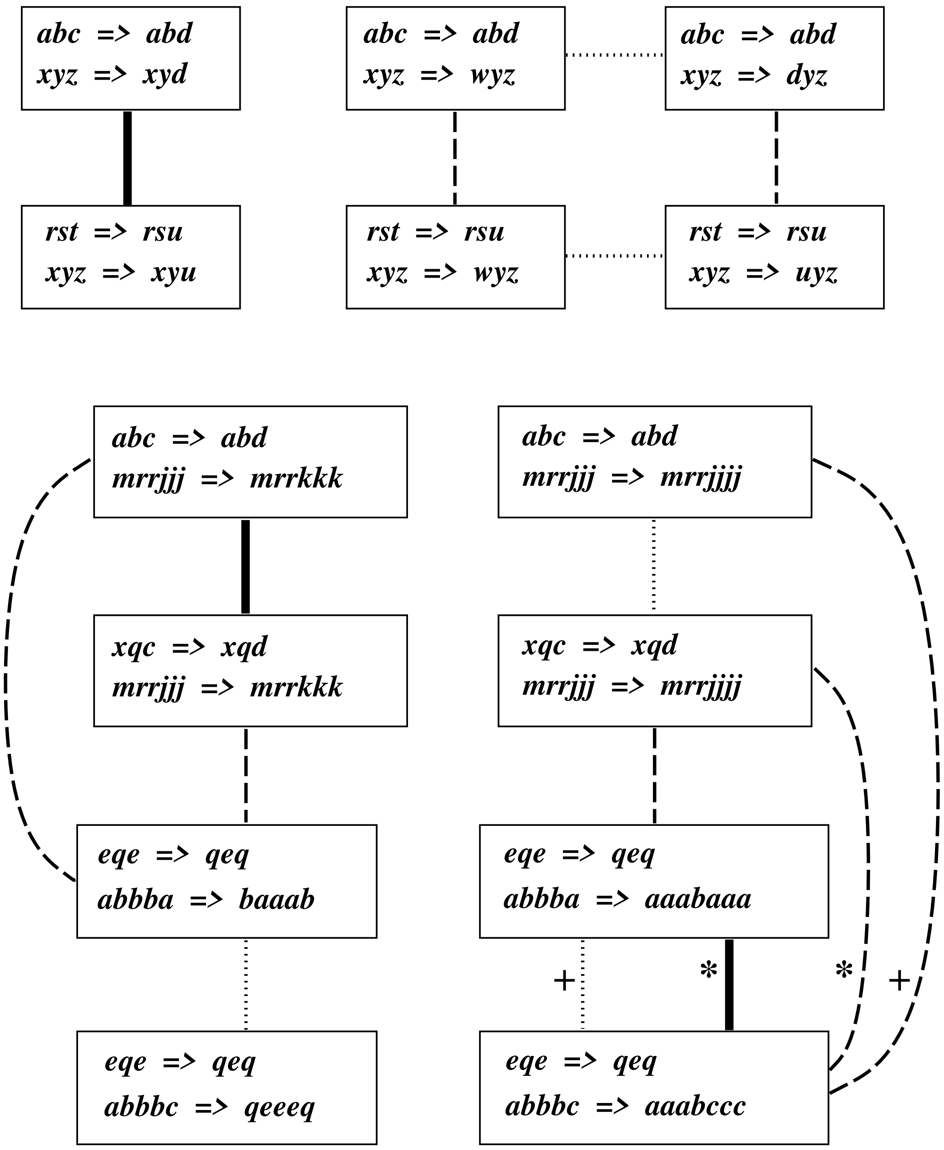


Figure 5.3: *Schematic diagram showing the different degrees to which various answers reminded Metacat of other answers it had encountered before. Heavy solid lines indicate strong reminding, dashed lines indicate a somewhat weaker degree of reminding, and dotted lines represent the weakest degree. Lines marked with * show remindings that occurred in the absence of a snag description, while those marked with + show remindings that occurred after the program had encountered a snag on its own.*

the program was given the problem *"**eqe** $\Rightarrow$ **qeq**; **abbbc** $\Rightarrow$ **?**"* to work on its own. In this particular run, it hit the snag a couple of times and then answered ***qeeeq***. The program was then given ***aaabccc*** again. This time, on account of the existing snag description, it was reminded only vaguely of ***aaabaaa***, and not at all of the answer ***mrrjjjj*** to the problem *"**xqc** $\Rightarrow$ **xqd**; **mrrjjj** $\Rightarrow$ **?**"*. Instead, it was reminded of the answer ***mrrjjjj*** to the problem *"**abc** $\Rightarrow$ **abd**; **mrrjjj** $\Rightarrow$ **?**"*, since seeing ***abbbc*** as *1–3–1* and ***mrrjjj*** as *1–2–3* both lead to stronger analogies in their respective problems, while seeing ***abbba*** as *1–3–1* does not. These remindings are marked with a plus sign (+) in the figure.

Once descriptions of all of these answers had accumulated in memory, the program was asked to compare selected pairs of answers in the manner described in section 4.7.3 of Chapter 4.[11] The following selection of commentary by the program, generated on the basis of the abstract answer descriptions stored in its memory, gives a sense of the kinds of parallels and distinctions between answers that Metacat is able to recognize.

Comparing answers to the same problem

In the following examples, Metacat compares different answers with respect to a single problem. This amounts to comparing answers "horizontally" in Figure 5.3.

abc/mrrkkk *vs.* ***abc/mrrjjjj***

> ***The answer mrrjjjj to the problem "abc -> abd, mrrjjj -> ?" is based in part on seeing abc and mrrjjj as groups of the same type by viewing one string in terms of letters and the other in terms of numbers. In contrast, in the case of the answer mrrkkk, the idea of seeing abc and mrrjjj as groups of the same type by viewing one string in terms of letters and the other in terms of numbers does not arise. All in all, I'd say mrrjjjj is the better answer, since it is based on a richer set of ideas.***

[11] "Asking" Metacat to compare two answers simply involves clicking the mouse on the appropriate answer description icons in the program's Episodic Memory window.

xqc/mrrkkk vs. *xqc/mrrjjjj*

The answer mrrjjjj to the problem "xqc –> xqd, mrrjjj –> ?" is based in part on viewing one of xqc and mrrjjj in terms of letters and the other in terms of numbers (although there is no good reason for doing so). In contrast, in the case of the answer mrrkkk, the idea of viewing one of xqc and mrrjjj in terms of letters and the other in terms of numbers does not arise. All in all, I'd say mrrkkk is the better answer, since it involves no unjustified ideas.

abc/xyd vs. *abc/wyz*

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.

rst/wyz vs. *rst/uyz*

The only essential difference between the answer uyz and the answer wyz to the problem "rst –> rsu, xyz –> ?" is that the change from rst to rsu is viewed in a more literal way for the answer uyz than it is in the case of wyz. Both answers rely on seeing two strings (rst and xyz in both cases) as symmetric predecessor and successor groups going in opposite directions. The answer uyz, however, seems incoherent to me, 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. All in all, I'd say wyz is the better answer, since it is more coherent.

eqe/aaabaaa vs. *eqe/baaab*

The answer aaabaaa to the problem "eqe –> qeq, abbba –> ?" is based in part on viewing one of eqe and abbba in terms of letters and the other in terms of numbers (although there is no good reason for doing so). In contrast, in the case of the answer baaab, the idea of viewing one of eqe and abbba in terms of letters and the other in terms of numbers does not arise. All in all, I'd say baaab is the better answer, since it involves no unjustified ideas.

eqe/qeeeq *vs.* ***eqe/aaabccc*** *(having encountered the snag before)*

> *The answer aaabccc to the problem "eqe -> qeq, abbbc -> ?" is based in part on seeing eqe and abbbc as groups of the same type by viewing one string in terms of letters and the other in terms of numbers (which 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). In contrast, in the case of the answer qeeeq, the idea of seeing eqe and abbbc as groups of the same type by viewing one string in terms of letters and the other in terms of numbers does not arise. Another key difference between the answers is that the change from eqe to qeq is viewed in a completely different way for the answer aaabccc than it is in the case of qeeeq. All in all, I'd say aaabccc is the better answer, since it is based on a richer set of ideas.*

Comparing answers to different problems within a single family

In the next set of examples, Metacat compares different answers to different problems within the same family. This amounts to comparing answers "vertically" in Figure 5.3.

abc/mrrkkk *vs.* ***xqc/mrrkkk***

> *The answer mrrkkk to the problem "abc -> abd, mrrjjj -> ?" is essentially the same as the answer mrrkkk to the problem "xqc -> xqd, mrrjjj -> ?". Both answers rely on seeing two strings (abc and mrrjjj in one case and xqc and mrrjjj in the other) as going in the same direction. Furthermore, the change from abc to abd is viewed in essentially the same way as the change from xqc to xqd. All in all, I'd say the first mrrkkk is very good and the second mrrkkk is pretty good.*

abc/mrrjjjj *vs.* ***xqc/mrrjjjj***

> *The answer mrrjjjj to the problem "abc -> abd, mrrjjj -> ?" is similar to the answer mrrjjjj to the problem "xqc -> xqd, mrrjjj -> ?", since both rely on seeing two strings (abc and mrrjjj in one case and xqc and mrrjjj 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 latter 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 former case with abc and mrrjjj. All in all, I'd say the first mrrjjjj is the better answer, since it involves no unjustified ideas.*

abc/xyd *vs.* ***rst/xyu***

> *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.*

abc/wyz *vs.* ***rst/wyz***

> *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.*

abc/dyz *vs.* ***rst/uyz***

> *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.*

eqe/qeeeq *vs.* ***eqe/baaab***

> *The only essential difference between the answer qeeeq to the problem "eqe –> qeq, abbbc –> ?" and the answer baaab to the problem "eqe –> qeq, abbba –> ?" is that the change from eqe to qeq is viewed in a completely different way for the answer qeeeq than it is in the case of baaab. Both answers rely on seeing two strings (eqe and abbbc in one case and eqe and abbba in the other) as going in the same direction. All in all, I'd say qeeeq is pretty bad and baaab is halfway decent.*

eqe/aaabaaa *vs.* ***eqe/aaabccc*** *(not having encountered the snag)*

> *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.*

eqe/aaabaaa *vs.* ***eqe/aaabccc*** *(having encountered the snag before)*

> *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.*

Comparing answers to problems in different families

In the last set of examples, Metacat compares answers to problems from different families. As before, this amounts to comparing answers "vertically" in Figure 5.3.

abc/mrrkkk *vs.* ***eqe/baaab***

> *The only essential difference between the answer mrrkkk to the problem "abc –> abd, mrrjjj –> ?" and the answer baaab to the problem "eqe –> qeq, abbba –> ?" is that the change from abc to abd is viewed in a completely different way for the answer mrrkkk than the change from eqe to qeq in the case of baaab. Both answers rely on seeing two strings (abc and mrrjjj in one case and eqe and abbba in the other) as going in the same direction. All in all, I'd say mrrkkk is very good and baaab is halfway decent.*

xqc/mrrkkk *vs.* ***eqe/baaab***

> *The only essential difference between the answer mrrkkk to the problem "xqc –> xqd, mrrjjj –> ?" and the answer baaab to the problem "eqe –> qeq, abbba –> ?" is that the change from xqc to xqd is viewed in a completely different way for the answer mrrkkk than the change from eqe to qeq in the case of baaab. Both answers rely on seeing two strings (xqc and mrrjjj in one case and eqe and abbba in the other) as going in the same direction. All in all, I'd say mrrkkk is pretty good and baaab is halfway decent.*

xqc/mrrjjjj *vs.* ***eqe/aaabaaa***

> *The only essential difference between the answer mrrjjjj to the problem "xqc –> xqd, mrrjjj –> ?" and the answer aaabaaa to the problem "eqe –> qeq, abbba –> ?" is that the change from xqc to xqd is viewed in a completely different way for the answer mrrjjjj than the change from eqe to qeq in the case of aaabaaa. Both answers rely on seeing two strings (xqc and mrrjjj in one case and eqe and abbba 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). All in all, I'd say mrrjjjj is pretty good and aaabaaa is halfway decent.*

abc/mrrjjjj *vs.* ***eqe/aaabccc*** *(having encountered the snag before)*

The only essential difference between the answer mrrjjjj to the problem "abc –> abd, mrrjjj –> ?" and the answer aaabccc to the problem "eqe –> qeq, abbbc –> ?" is that the change from abc to abd is viewed in a completely different way for the answer mrrjjjj than the change from eqe to qeq in the case of aaabccc. Both answers rely on seeing two strings (abc and mrrjjj in one case and eqe and abbbc in the other) as groups of the same type going in the same direction, and on viewing one of the strings in terms of letters and the other in terms of numbers. All in all, I'd say mrrjjjj is very good and aaabccc is pretty good.

xqc/mrrjjjj *vs.* ***eqe/aaabccc*** *(not having encountered the snag)*

The only essential difference between the answer mrrjjjj to the problem "xqc –> xqd, mrrjjj –> ?" and the answer aaabccc to the problem "eqe –> qeq, abbbc –> ?" is that the change from xqc to xqd is viewed in a completely different way for the answer mrrjjjj than the change from eqe to qeq in the case of aaabccc. Both answers rely on seeing two strings (xqc and mrrjjj 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). All in all, I'd say they're both pretty good answers.

abc/wyz *vs.* ***abc/mrrjjjj***

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 mrrjjjj to the problem "abc –> abd, mrrjjj –> ?" is based on seeing abc and mrrjjj as groups of the same type going in the same direction, and on viewing one of the strings in terms of letters and the other in terms of numbers. In mrrjjjj's case, the idea of seeing alphabetic–position symmetry between abc and mrrjjj does not arise. In wyz's case, the idea of viewing one of abc and xyz in terms of letters and the other in terms of numbers does not arise. All in all, I'd say they're both very good answers.

5.3 Problems with the Model

The sample runs presented in the last section demonstrated most of the capabilities of Metacat's self-watching mechanisms, and were chosen because they serve to illustrate the current strengths of the program. In contrast, this section discusses some of the weaknesses that remain in the current version of Metacat. The weaknesses to be discussed here all pertain to various shortcomings of currently existing mechanisms, rather than to the absence of mechanisms that would be desirable for the program to have. (The next chapter outlines various capabilities that would be desirable to incorporate into the program in any future work on the project.)

5.3.1 Implausible rules

One lingering problem with the program is its tendency to create overly complicated and cumbersome rules for describing string changes. Although almost all of the rules appearing in the earlier sample runs are quite reasonable (the exception being perhaps the rule appearing in Panel *6-a*), Metacat all too often comes up with rules that describe string changes in very bizarre ways. Although these rules technically "work", they are quite implausible, in the sense that it is very unlikely that a human would even think of describing the strings in question in such a convoluted fashion (except perhaps as a joke).

As an example, consider the problem *"**eeqee** ⇒ **qeeq**; **xxixx** ⇒ **?**"*. A natural way to describe the ***eeqee*** ⇒ ***qeeq*** change is with the rule *Swap letter-categories and lengths of all objects in string* (assuming that the ***ee*** groups are perceived as single units). Metacat can discover this rule without too much difficulty, but it also typically comes up with a large number of other, quite outlandish rules for the ***eeqee*** ⇒ ***qeeq*** change when given this problem. A few such rules are shown in Figures 5.4 and 5.5, along with the answers they yield. (For clarity, the corresponding bottom rules are

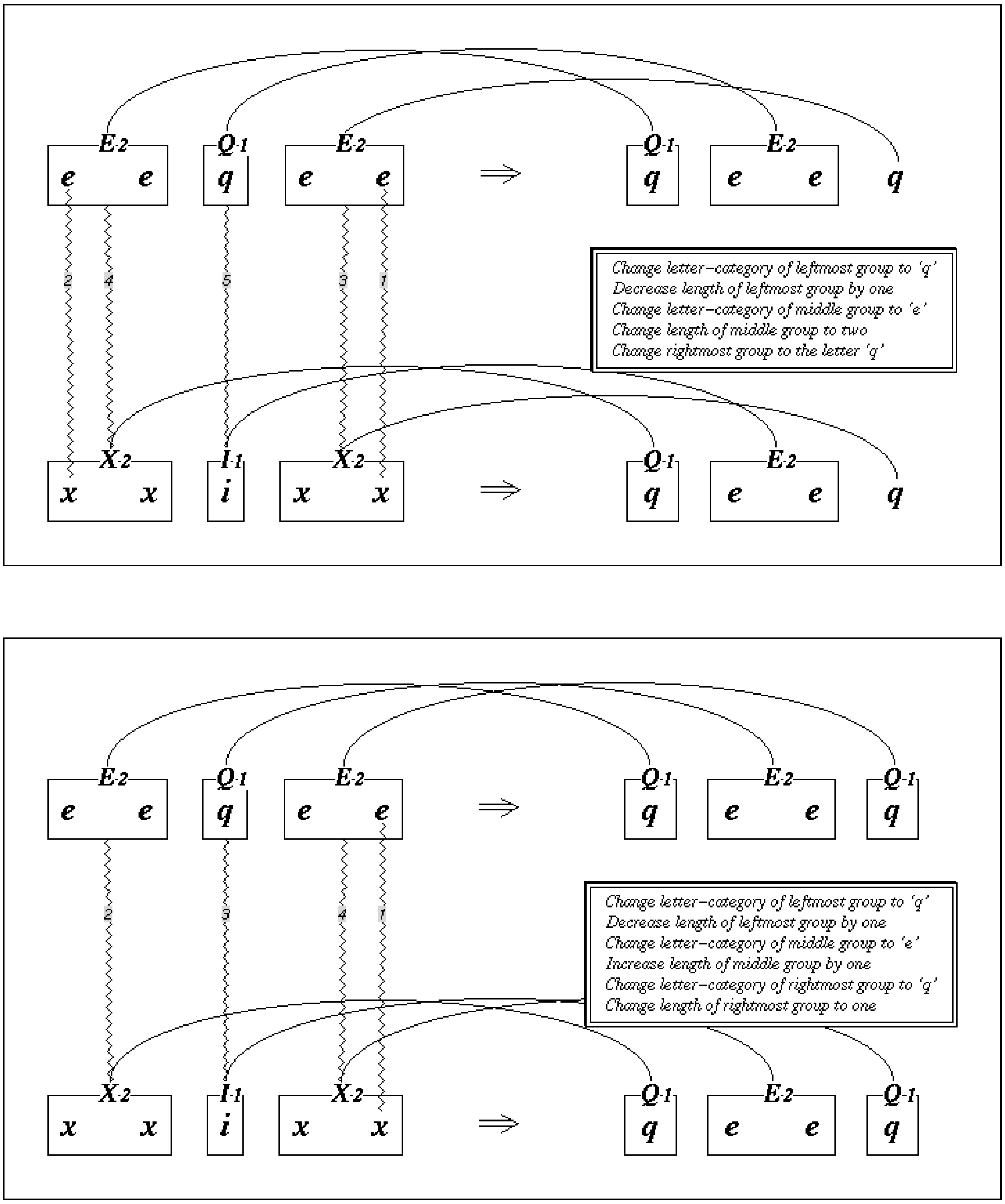
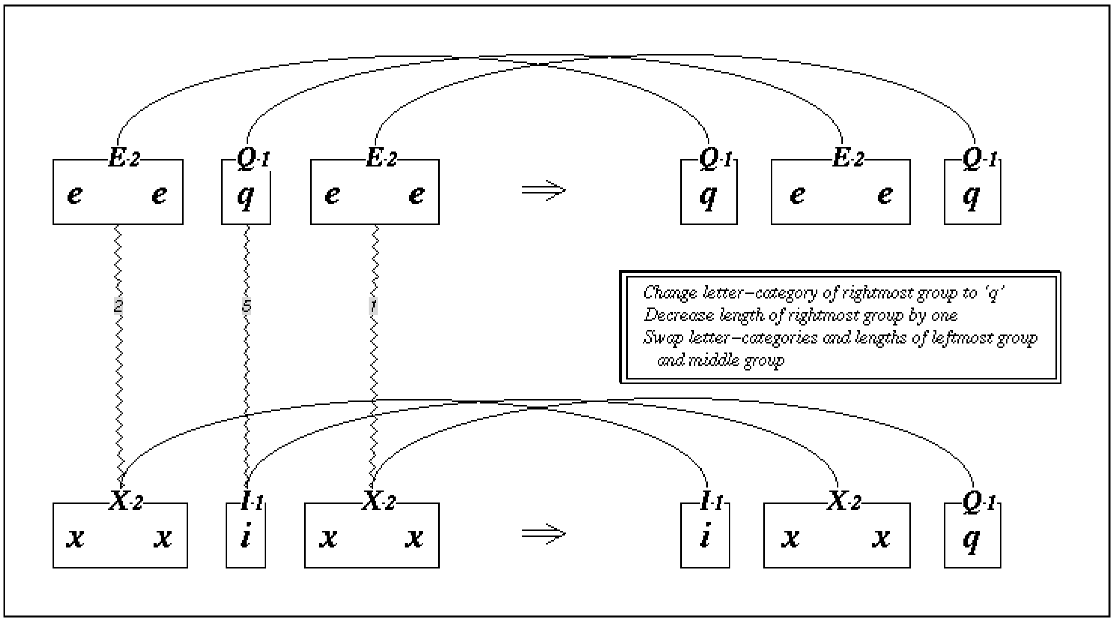


Figure 5.4: *Two convoluted rules created by the program for describing the* ***eeqee*** ⇒ ***qeeq*** *change during runs of the problem "****eeqee*** ⇒ ***qeeq; xxixx*** ⇒ ***?****". In both cases, the translated rule (not shown) is identical to the top rule.*



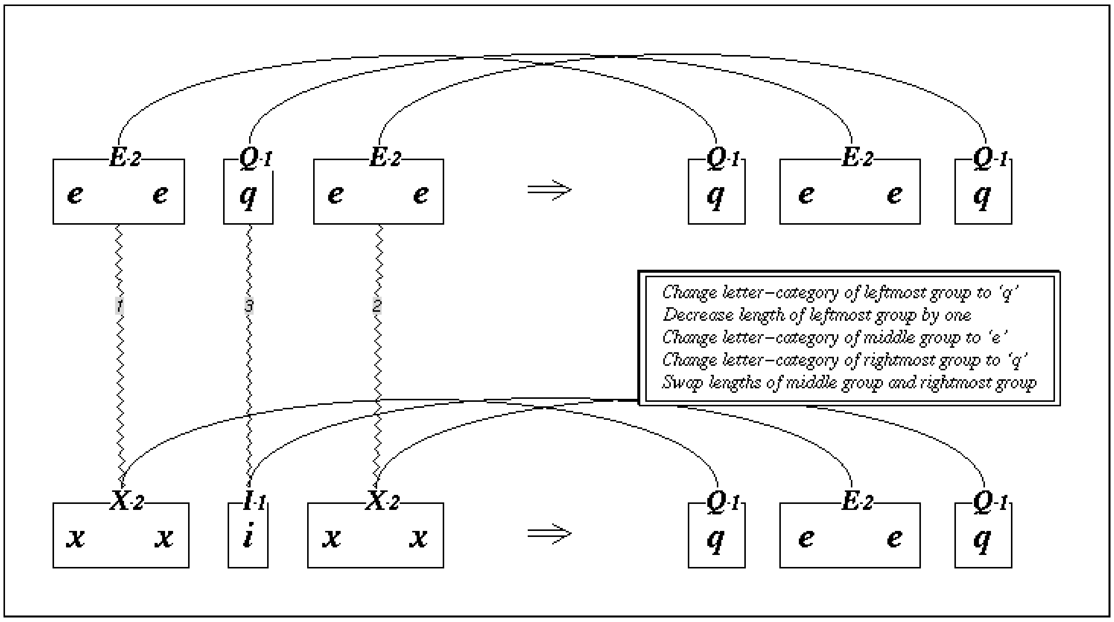


Figure 5.5: *Another pair of convoluted rules created by the program for describing the* ***eeqee*** ⇒ ***qeeq*** *change in the problem "****eeqee*** ⇒ ***qeeq; xxixx*** ⇒ ***?****".*

not shown, since they are identical to the top rules in each case, due to the absence of any vertical slippages between ***eeqee*** and ***xxixx***.)

The first rule shown in Figure 5.4 specifies a rather haphazard mishmash of changes to individual components of the string ***eeqee***. According to this rule, the leftmost ***ee*** group and the middle ***q*** group independently change their letter-categories to q and e, respectively. Changes to their lengths, however, are described using different levels of abstraction: as decreasing by one in the case of ***ee***, and as changing literally to two in the case of ***q***. In contrast, the rightmost ***ee*** group is described as simply changing into the letter ***q***. The second rule spells out each individual letter-category and length change explicitly, but does so in an inconsistent manner in the case of the length changes, since the rightmost ***ee*** group's length is described literally as changing to one, while the lengths of the other groups are described as either increasing or decreasing.

The rules shown in Figure 5.5 are even more incongruous, since each one describes the change to ***eeqee*** partially in terms of swapping attributes of string components, and partially in terms of individual changes to components. In the case of the second rule, the changes to the letter-categories of the middle ***q*** group and the rightmost ***ee*** group are not even described in the same way as the changes to the groups' lengths (*i.e.*, the letter-category changes are described individually, while the length changes are described as a swap).

Several other "monster rules" typically created by the program during runs of the problem *"**eeqee** ⇒ **qeeq**; **xxixx** ⇒ **?**"* are shown below:

- *Change letter-category of leftmost group to 'q'*
  Change length of leftmost group to one
  Change letter-category of middle group to 'e'
  Change length of middle group to two
  Change rightmost group to the letter 'q'

- *Change leftmost group to the letter 'q'*
  Change letter-category of middle group to 'e'
  Increase length of middle group by one
  Change letter-category of rightmost group to 'q'
  Decrease length of rightmost group by one

- *Change letter-category of leftmost group to 'q'*
  Decrease length of leftmost group by one
  Change letter-category of middle group to 'e'
  Increase length of middle group by one
  Change letter-category of rightmost group to 'q'
  Decrease length of rightmost group by one

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

- *Change letter-category of leftmost group to 'q'*
  Change letter-category of middle group to 'e'
  Change letter-category of rightmost group to 'q'
  Decrease length of rightmost group by one
  Swap lengths of leftmost group and middle group

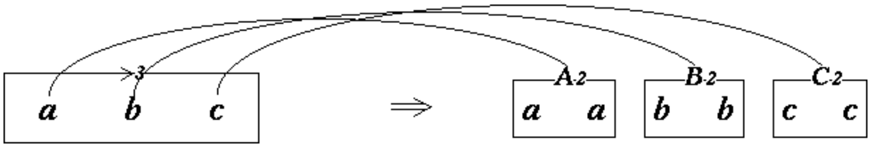
- *Decrease length of leftmost group by one*
  Increase length of middle group by one
  Decrease length of rightmost group by one
  Swap letter-categories of all objects in string

- *Change letter-category of leftmost group to 'q'*
  Decrease length of leftmost group by one
  Increase length of middle group by one
  Decrease length of rightmost group by one
  Swap letter-categories of middle group and rightmost group

From a purely technical standpoint, each of these rules represents a valid way of describing the ***eeqee*** $\Rightarrow$ ***qeeq*** change, since each one produces the string ***qeeq*** as required, when applied to ***eeqee***. People, however, are not likely to perceive ***eeqee*** as changing in such disjointed and convoluted ways. Indeed, any tendency for a person to perceive situations in the world in a comparably disconnected and inconsistent manner would probably constitute grounds for some concern. Unfortunately, the tendency for Metacat to do so is uncomfortably strong, and consequently represents a serious shortcoming of the model. Although the program—to its credit—judges most of these rules to be of low quality, its propensity for creating such unwieldy rules in the first place is not very psychologically realistic, since most of these rules would not even *occur* to people.

In a way, the fact that seemingly simple string changes such as ***eeqee*** $\Rightarrow$ ***qeeq*** can potentially be described in so many different ways attests to the inherent subtlety and complexity of the letter-string microworld. (In fact, the rules shown above for ***eeqee*** $\Rightarrow$ ***qeeq*** represent only a small fraction of the myriad possible ways in which this particular string change can be described.) Furthermore, the fact that the more "exotic" ways of describing changes to strings rarely occur to people attests to the strong tendency of human perception to automatically filter out incoherent ways of perceiving situations. On the other hand, the *potential* for perceiving situations in unorthodox ways plays a critical role in human creativity. Accordingly, for Metacat to be a more faithful model of human cognition, its "perceptual filters" need to be strengthened so that it will be less inclined to describe string changes in bizarre ways, while still retaining the potential for doing so.

Another problem with Metacat's rule mechanisms is that sometimes two rules that should really be considered to be equivalent are regarded as distinct by the program. In particular, this tends to happen in the case of rules that involve letter-to-group or group-to-letter changes. For example, Figure 5.6 shows two rules that the



Increase length of leftmost letter by one
Increase length of middle letter by one
Increase length of rightmost letter by one

```
((intrinsic ((letter String-Position leftmost))
            ((object Length successor)
             (object Object-Category group)))
 (intrinsic ((letter String-Position middle))
            ((object Length successor)
             (object Object-Category group)))
 (intrinsic ((letter String-Position rightmost))
            ((object Length successor)
             (object Object-Category group))))
```

Increase length of leftmost letter by one
Increase length of middle letter by one
Increase length of rightmost letter by one
Change all objects in whole group to groups

```
((intrinsic ((letter String-Position leftmost))
            ((object Length successor)))
 (intrinsic ((letter String-Position middle))
            ((object Length successor)))
 (intrinsic ((letter String-Position rightmost))
            ((object Length successor)))
 (intrinsic ((group String-Position whole))
            ((components Object-Category group))))
```

Figure 5.6: *Two structurally-distinct but essentially equivalent rules created by Metacat for describing the change* ***abc*** *⇒* ***aabbcc****. As far as the program is concerned, these rules represent utterly different ways of looking at* ***abc*** *⇒* ***aabbcc****.*

program created to describe the string change ***abc*** ⇒ ***aabbcc***. (The internal structure of each rule appears beneath its English rendition.) The only difference between these rules is that the first rule describes the letters ***a***, ***b***, and ***c*** as changing on a purely individual basis, while the second involves a mixture of individual changes (in the case of *Length*) and collective changes (in the case of *Object-Category*). However, the *Object-Category* changes are essentially redundant, since increasing the length of a letter automatically implies changing it to a group. In other words, both of these rules convey essentially the same information, and should thus be regarded as equivalent. Unfortunately, Metacat's rule mechanisms are not clever enough to recognize this type of rule equivalence. As a consequence, the program may end up generating a slew of identical answers for a problem, all based on structurally distinct—but essentially equivalent—rules. This problem is further compounded by the tendency of the program to create many different variations of a single rule, based on minor permutations of its constituent concepts, as was illustrated by the proliferation of the ***eeqee*** ⇒ ***qeeq*** rules shown earlier.

5.3.2 Poor thematic characterizations of answers

Another weakness of the program has to do with its mechanisms for creating descriptions of answers in terms of themes. Sometimes the resulting thematic characterizations do not accurately reflect the answers they are intended to describe. This may in turn lead the program to rather peculiar conclusions about the similarities and differences between various answers.

For instance, consider the problem *"**aabc** ⇒ **aabd**; **ijkk** ⇒ **?**"*. One possible answer is ***ijll***, based on seeing ***aabc*** and ***ijkk*** as going in the same direction, with the rightmost ***kk*** group of ***ijkk*** corresponding to the rightmost letter ***c*** of ***aabc***. This way of looking at the problem ignores the fact that since ***aabc*** and ***ijkk*** contain doubled letters, they share a strong underlying similarity. Taking this similarity into account

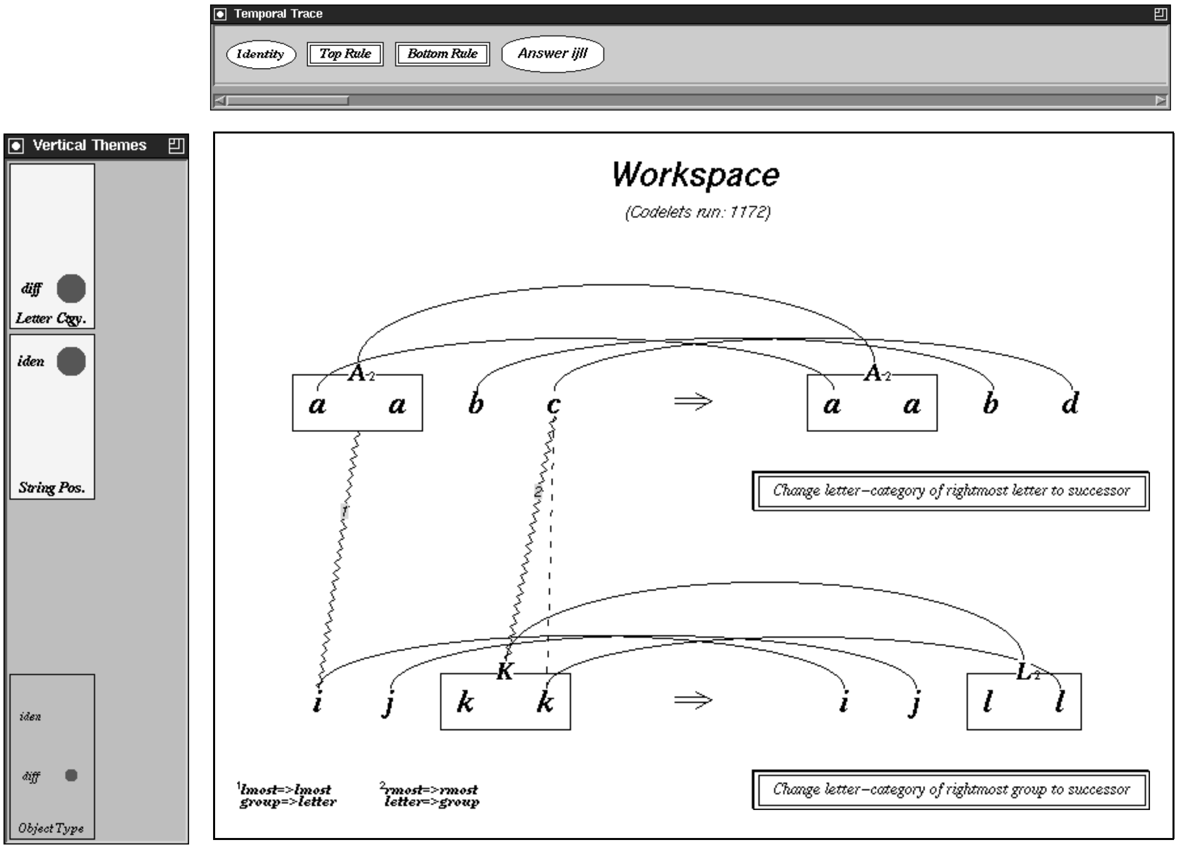


Figure 5.7: *The result of a justification run for the answer **ijll**.*

suggests the answer ***hjkk***, which arises from viewing ***aabc*** and ***ijkk*** as "anchored" at opposite ends by doubled letters, with the letter at the other end changing "by one" (the ***c*** to its successor, and, symmetrically, the ***i*** to its predecessor). This answer is considerably more elegant than ***ijll***.

Unfortunately, Metacat may fail to recognize the very different character of these two answers, even after successfully making sense of each one individually. As an illustration of this, Figure 5.7 shows the final state of Metacat's Workspace after a justification run for the answer ***ijll***. The final activations of vertical themes in the Themespace are also shown. The program includes the vertical *String-Position: identity* theme in its answer description for ***ijll***, due to the strong activation of this theme by the vertical bridges ***aa–i*** and ***c–kk***.[12]

On the other hand, Figure 5.8 shows a justification run for the answer ***hjkk***. The top of the figure shows the Workspace after 520 codelets have run. Up to this point, the program has not yet noticed the similarity between ***aa*** and ***kk*** (although it is flirting with this idea). The vertical *String-Position: identity* theme is thus highly active, on account of the ***a–i*** and ***c–k*** bridges. Soon afterwards, however, at time step 541, these bridges are broken and replaced by a diagonal ***aa–kk*** bridge, depriving the *String-Position: identity* theme of its support. Consequently, the activation of this theme begins to diminish, while that of the competing *String-Position: opposite* theme begins to rise. Less than 200 codelets later, at time step 730, a symmetric ***c–i*** bridge is built, leading quickly to a successful justification of ***hjkk*** at time step 733 (see the bottom of Figure 5.8).

As can be seen from the figure, however, the run ended before the vertical *String-Position: opposite* theme could attain dominance over the *String-Position: identity* theme. Furthermore, neither the ***aa–kk*** bridge's *leftmost ⇒ rightmost* slippage nor the ***c–i*** bridge's *rightmost ⇒ leftmost* slippage appears in the Temporal Trace, because neither slippage was made under thematic pressure, and neither one involves an important group (as judged by the program). In general, in the absence of a dominant *String-Position* theme in the Themespace or an explicit *String-Position* slippage in the Trace, Metacat assumes (incorrectly, in this case) that the initial and target strings map onto each other in a straightforward, parallel fashion. It therefore includes a *String-Position: identity* theme in its answer description for ***hjkk***. Consequently, after justifying ***hjkk***, the program reports being "strongly reminded" of the earlier answer ***ijll***, since ***ijll***'s answer description also includes a *String-Position: identity* theme (as well as exactly the same rule). Indeed, when asked to compare these two answers, the program judges ***hjkk*** to be "essentially the same" as ***ijll*** (see Figure 5.9).

[12] See section 4.7.1 of Chapter 4 for a discussion of how Metacat creates answer descriptions.

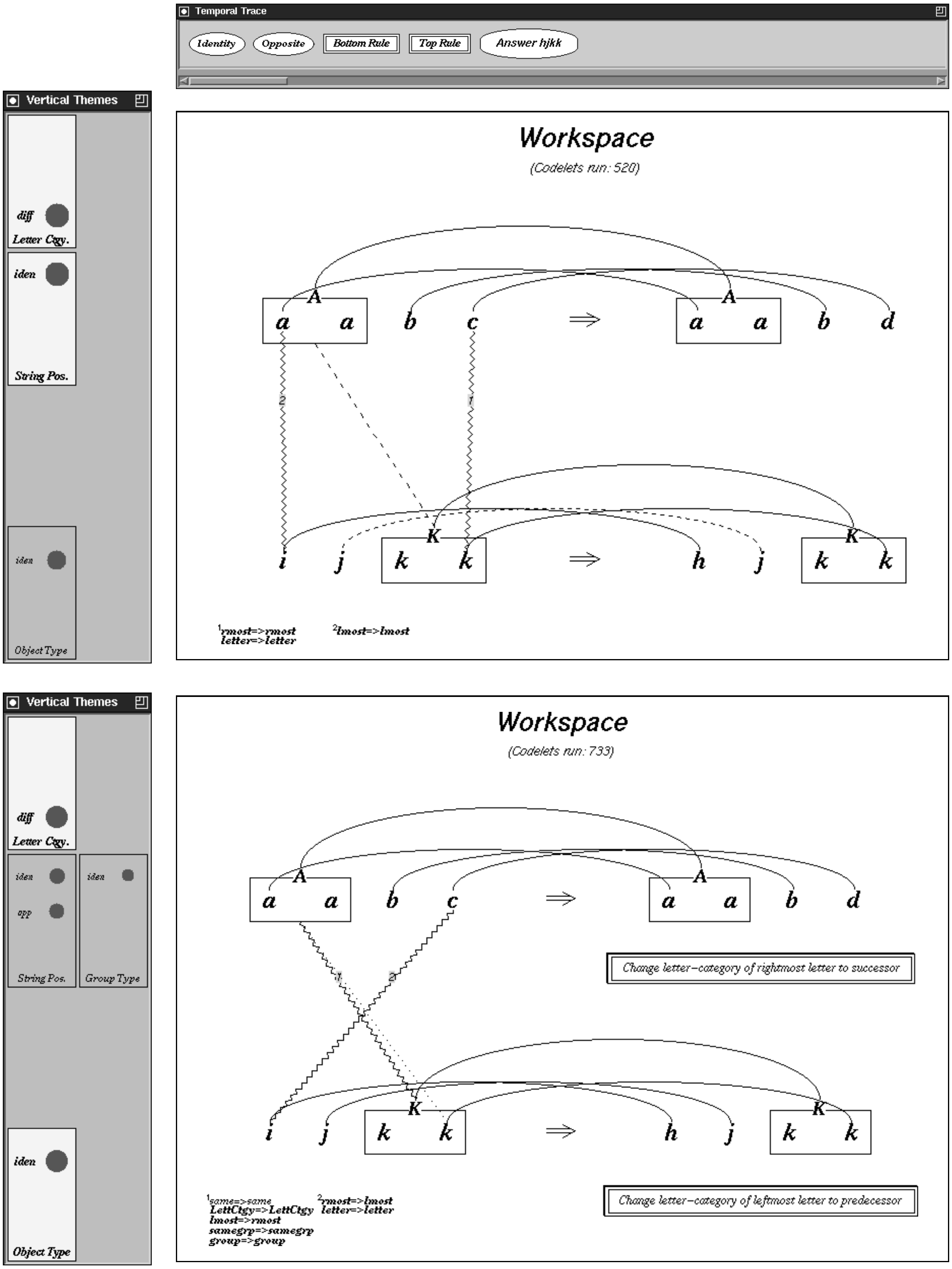


Figure 5.8: *A justification run for **hjkk** resulting in a misleading answer description.*

> ***The answer hjkk is essentially the same as the answer ijll to the problem "aabc -> aabd, ijkk -> ?". Both answers rely on seeing two strings (aabc and ijkk in both cases) as going in the same direction. Furthermore, the change from aabc to aabd is viewed in essentially the same way in both cases. All in all, I'd say they're both very good answers.***

Figure 5.9: *Metacat's assessment of the answers **hjkk** and **ijll**.*

Needless to say, this is a rather odd claim to make.

Another amusing case in which Metacat arrives at questionable conclusions about the relative merits of two answers it has discovered involves the problem *"**abc** ⇒ **abd**; **xyz** ⇒ **?**"*. Figure 5.10 shows the Temporal Trace for the final portion of a run in which the program discovered the answer ***wyz***. The answer description created for ***wyz*** is also shown, including the answer's associated vertical themes. In this run, Metacat hit the usual ***z***-snag several times and then negatively clamped the vertical theme *String-Position: identity*, which resulted in the creation of a symmetric mapping between ***abc*** and ***xyz***. This subsequently led to the discovery of the answer ***wyz***. However, the program failed to notice the alphabetic-position symmetry between ***a*** and ***z*** (as can be seen from the absence of a corresponding *first* ⇒ *last* slippage in the Trace). Thus, no *Alphabetic-Position: opposite* theme was included in ***wyz***'s answer description.

In contrast, Figure 5.11 shows the final portion of a different run of the same problem, in which the answer ***yyz*** was found. (This run is actually a continuation of the run shown in Figure 4.12 of Chapter 4.) In this run, after hitting the snag several times, the program negatively clamped the vertical *String-Position: identity* theme, which, as before, led to the creation of a crosswise mapping between ***abc*** and ***xyz***. However, in this run, the program happened to make the *first* ⇒ *last* slippage between ***a*** and ***z***, unlike in the previous run. (In fact, this slippage was made twice, because the ***a***–***z*** bridge got broken and then rebuilt.) Thus the program included an

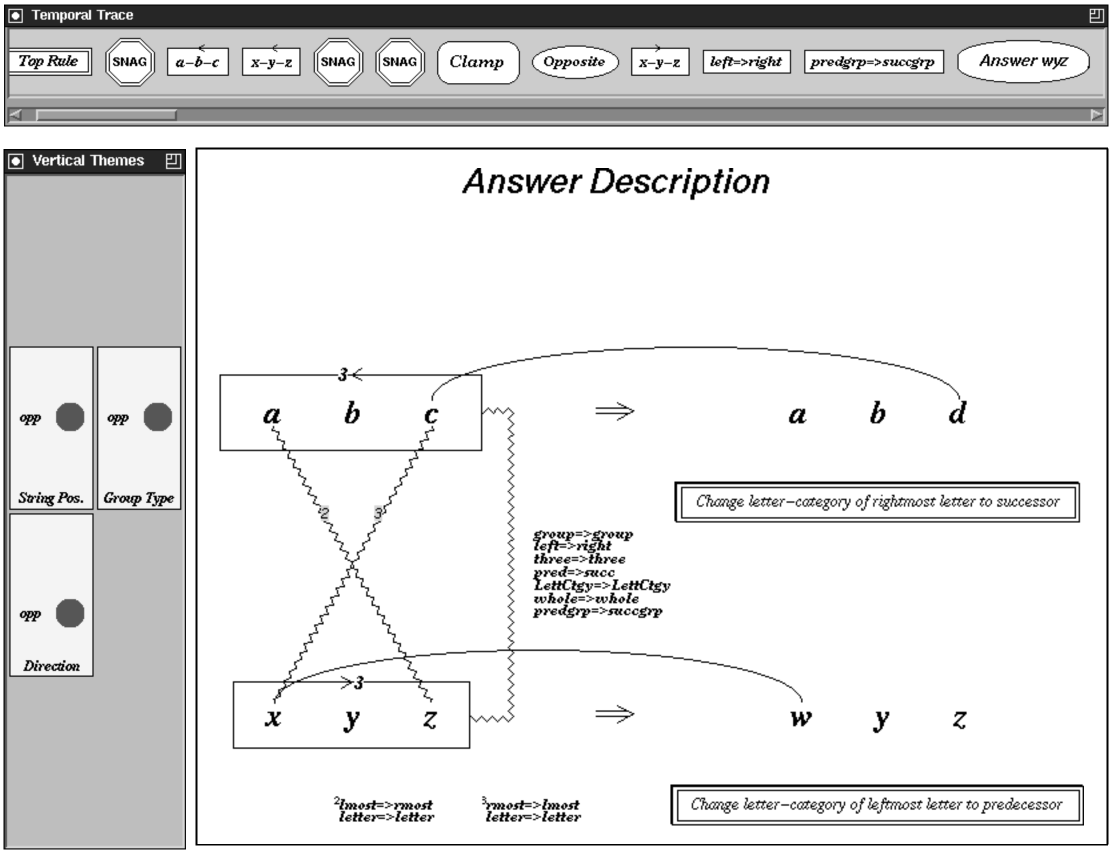


Figure 5.10: *The answer description created for* ***wyz****.*

Alphabetic-Position: opposite theme in its answer description for ***yyz***, on account of the *first* ⇒ *last* slippages appearing in the Trace.

Figure 5.12 shows Metacat's resulting commentary on the two answers ***yyz*** and ***wyz***. In this case, as can be seen, the program prefers the answer ***yyz***, on account of the extra *Alphabetic-Position: opposite* theme included in ***yyz***'s answer description. However, this is somewhat ironic, considering that even Copycat strongly prefers the answer ***wyz*** (to which it assigns, on average, a final temperature of 14, as opposed to 44 for ***yyz*** [Mitchell, 1993]), as do many people. Of course, on other runs of this problem, Metacat, too, may express the opposite preference—as long as the *first* ⇒ *last* slippage is made for ***wyz*** but not for ***yyz***. (On the other hand, if this

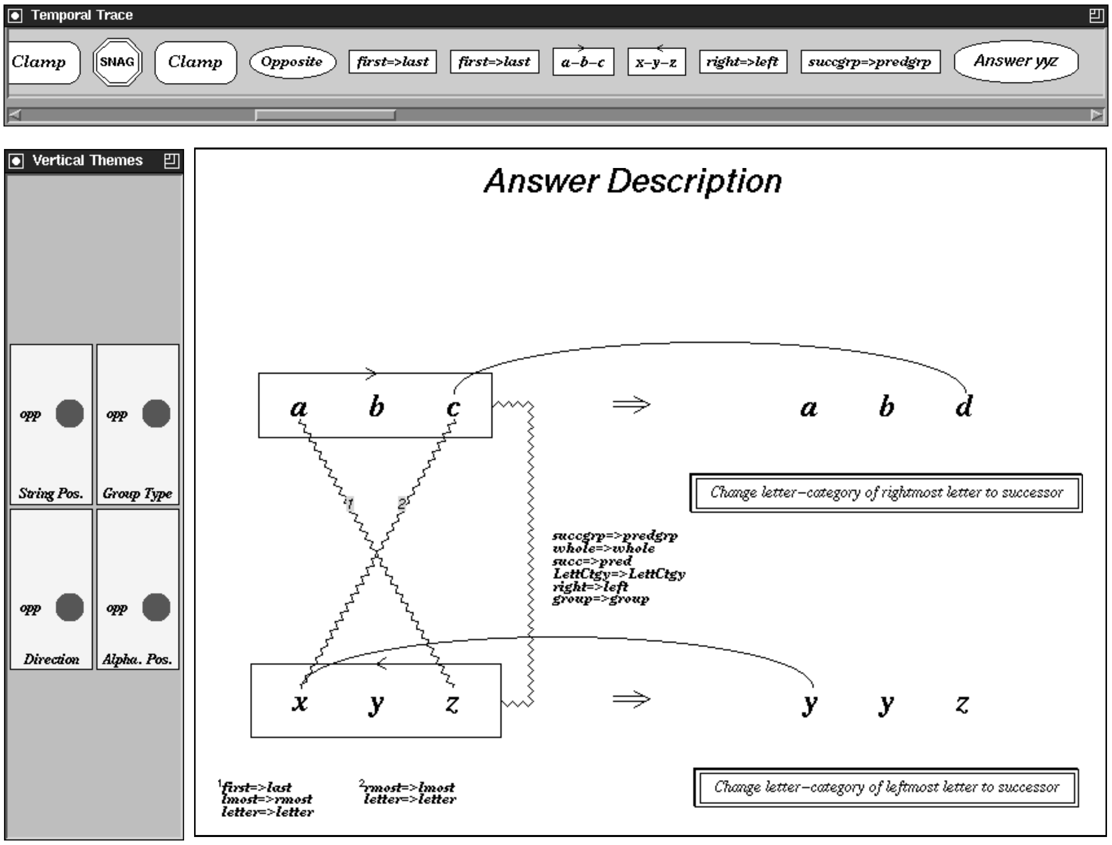


Figure 5.11: *The answer description created for* ***yyz****.*

slippage happens to be made for *both* answers, then the program proclaims ***wyz*** and ***yyz*** to be "essentially the same".)

In any case, this example underscores both the fragility and the shallowness of Metacat's insight into its answers. To be sure, Metacat has more insight into its answers than does Copycat, because it can at least explicitly point to certain ideas as being of particular importance, such as the idea of alphabetic-position symmetry in the present example. But this insight depends critically on the program's having included the appropriate themes in its descriptions of its answers. Furthermore, even if the appropriate themes are included, the program may still fail to recognize many important aspects of an answer. (For instance, nowhere in its commentary on ***yyz***

> *The answer yyz 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, the idea of seeing alphabetic-position symmetry between abc and xyz does not arise. All in all, I'd say yyz is the better answer, since it is based on a richer set of ideas.*

Figure 5.12: *Metacat's assessment of the answers **yyz** and **wyz**.*

does Metacat point out the fundamental inconsistency of seeing ***abc*** and ***xyz*** as alphabetically-symmetric *opposites* while still changing the letter ***x*** to its *successor*, instead of to its predecessor.)

Nevertheless, the fact that the program can—at least some of the time—perceive quite subtle parallels and distinctions between different analogies represents a significant step beyond the perceptual abilities of Copycat, even though, clearly, much room for improvement still exists. All in all, I'd say Metacat is the better program, since it is based on a richer set of mechanisms.

CHAPTER SIX

Conclusion

This chapter summarizes the work on the Metacat project presented in this dissertation, and discusses possible directions in which future work on the project might proceed. The first section reviews the major ideas presented in each chapter.

6.1 Summary

Chapter 1 provided background on the Copycat model of analogy-making and high-level perception, out of which the Metacat project grew. Since Metacat is an extension of Copycat, this background was needed in order to understand the subsequent work on Metacat presented in later chapters. The chapter began with a general discussion of the central ideas motivating the development of Copycat, including the notion of *high-level perception* (*i.e.*, the ability to perceive situations in terms of abstract *concepts*), and the notion of *conceptual fluidity* (*i.e.*, the highly flexible and context-sensitive nature of these concepts), which gives rise automatically to an ability to perceive analogies through the mechanism of *conceptual slippage*. Copycat's idealized letter-string microdomain was then described, and the crucial idea of the domain's universality was stressed—that is, the fact that the program treats letter-strings simply as abstract configurations of objects and relationships, without knowing anything about letters *per se*.

This was followed by a discussion of the principal components of Copycat's architecture, including the Workspace (where the letter-strings reside and where perceptual activity consequently takes place), the Slipnet (where the program's repertoire of concepts about its letter-string microworld are stored), and the Coderack (where codelets wait to be chosen to run). The important role played by temperature in guiding the program's stochastic processing mechanisms was also discussed. Together, these mechanisms give rise to the *parallel terraced scan*, which allows Copycat to simultaneously explore many potential pathways through its search space at different speeds, according to each pathway's estimated degree of promise.

Chapter 2 put the work on Copycat into perspective by first describing several other projects closely related to Copycat, all of which use the same general stochastic codelet architecture. This was followed by a discussion of Copycat's principal weaknesses as a realistic model of human cognition. Several weaknesses were identified, most of which stem from the program's lack of insight into what it is doing when it solves analogy problems—on account of its inability to explicitly remember anything that happens during a run.

The discussion of Copycat's weaknesses was followed by an outline of the objectives of the Metacat project, most of which are concerned with remedying the weaknesses just described. These objectives include making the program sensitive to patterns in its own processing (especially to repetitive patterns of behavior) by developing mechanisms to support an in-depth capacity for *self-watching*; giving the program the ability to remember its answers and to be reminded of other answers that it has encountered before; enriching the information associated with answers so that they can be compared and contrasted in an insightful manner; giving the program the ability to make sense of an answer provided to it by "working backwards"; and relaxing Copycat's rigid constraints on rules so that a wider class of analogy problems can be handled.

The main architectural components of Metacat were described next. The three new components of the architecture (not present in Copycat) are summarized below:

- *The Themespace* contains structures that explicitly represent ideas that play a key role in the program's current interpretation of an analogy problem. These structures, called *themes*, are composed of Slipnet concepts, and have time-varying activation levels that change as the program explores different ways of looking at its letter-strings. Under certain conditions, themes can exert strong top-down pressure on Metacat's stochastic processing mechanisms, forcing the program to look at the strings in a particular way. Furthermore, themes serve as the basis for comparing and contrasting different answers, since they are the principal constituents of Metacat's abstract descriptions of its answers.

- *The Temporal Trace* contains structures that explicitly represent important processing events that occur during a run. Such events include the creation of important Workspace structures, the activation of deep Slipnet concepts, the occurrence of slippages, the discovery of new answers, running into snags, and explicitly focusing on particular ideas.

- *The Episodic Memory* stores abstract descriptions of answers, and consequently serves as the program's long-term repository for its problem-solving experience.

Three extended examples were then presented, in order to more clearly illustrate the ways in which these architectural components enable Metacat to watch (and to respond to) its own behavior, to "work backwards" from a given answer to an interpretation that makes sense for the answer, and to compare and contrast different answers on the basis of themes. Finally, Metacat's relation to other work in AI and cognitive science was discussed.

Chapter 3 presented an in-depth discussion of Metacat's generalized rule-building mechanisms, and gave many examples of rules that the program is now able to build (but which were not possible in Copycat). The internal structure of rules was also described, as well as the three rule-quality measures of *uniformity*, *abstractness*, and *succinctness*. This was followed by a detailed description of the program's method for creating new rules from the bridges between strings, in which regularities among the concept-mappings underlying the bridges are transformed into a set of "intrinsic" and "extrinsic" *rule clauses* that describe the differences between the strings. The nondeterministic nature of Metacat's rule-translation process was described next, together with the notion of *coattail slippages*—whereby a slippage involving a particular relationship between one pair of concepts may occasionally induce slippages involving the same relationship between other pairs of concepts. A few other refinements to mechanisms inherited from Copycat were also discussed.

Chapter 4—the heart of the dissertation—began by describing the Themespace in detail, and the ways in which themes can control the high-level behavior of Metacat by exerting strong top-down pressure on the program's stochastic processing mechanisms. Various examples of *patterns* were presented next, including *theme-patterns*, *concept-patterns*, and *codelet-patterns*. The following section illustrated how top-down pressure exerted by clamping different types of patterns enables the program to size up answers provided to it, by "working backwards" in *justify mode*. This was followed by a description of the Temporal Trace and the types of processing events that can be recorded therein.

The next section tied together themes, pattern-clamping, and the Temporal Trace by explaining how these mechanisms make it possible for Metacat to monitor its own behavior—at a highly chunked level of description—and to respond to this behavior in appropriate ways, such as by breaking out of unproductive, mindlessly repetitive patterns of behavior via *jootsing*. Next, examples of the program's ability to describe its answers and its behavior in English were presented. At the same time, however, the canned nature of much of the program's linguistic output was carefully stressed.

The last section discussed Metacat's Episodic Memory. The nature of the descriptions created by the program to characterize its answers—as well as the snags that it encounters in searching for answers—was first described. These descriptions, stored in memory, enable the program to recognize subtle parallels and distinctions between different answers on the basis of the themes included in the descriptions. A detailed example showing how Metacat constructs English-language summaries of the similarities and differences between answers was then presented. Finally, the ability of the program to be reminded of one answer by another according to the similarity of the themes associated with the answers was discussed.

Chapter 5 presented in detail a number of complete sample runs of Metacat on several families of analogy problems, in order to demonstrate more clearly the mechanisms discussed in Chapters 3 and 4. In addition, many examples were given of the output generated by the program when comparing different analogies from these families. In contrast, the last section of the chapter presented examples that illustrated a number of weaknesses of the current version of the program. These weaknesses include the tendency of Metacat to create overly-complicated and implausible rules for describing string changes, as well as its failure from time to time to characterize answers in terms of the appropriate themes, which may lead the program to unwarranted conclusions about the similarities or differences between answers.

6.2 Contributions and Future Work

The research presented in this dissertation represents another small step down a very long road leading toward a deeper understanding of the nature of *concepts*, and of the pivotal role they play in human cognition. As with Metacat's forerunners along this road, the bedrock assumption underlying this work is that only by understanding the nature of concepts in a genuine and deep way will other aspects of cognition—including analogy, memory organization, reminding, and self-awareness—come within reach of understanding. Indeed, it is fair to say that understanding concepts is the central problem of cognitive science and artificial intelligence.

In particular, this focus on concepts is what most clearly distinguishes the approach to analogy taken by Metacat and Copycat from approaches developed by other researchers—such as the ACME model of Holyoak and Thagard [Holyoak and Thagard, 1989], the SME model of Falkenhainer, Forbus, and Gentner [Falkenhainer et al., 1990], or the derivational analogy approach of Carbonell [Carbonell, 1986]. Likewise, Metacat's approach to the issues of memory organization and reminding differs from other approaches—such as the ARCS model of Holyoak and Thagard [Thagard et al., 1990], the MAC/FAC model of Forbus and Gentner [Forbus et al., 1995], or the many CBR models of memory and learning descended from Schank's theories of memory organization [Leake, 1996; Schank, 1982]—on account of the former's commitment to *taking concepts seriously.*

Copycat, too, takes concepts seriously, but whereas Copycat is concerned with elucidating the ways in which concepts interact with the perception of similarity between potentially disparate situations, Metacat is concerned with the ways in which concepts interact with *self-perception.* Both types of perception are crucial to cognition, but the goal of a full accounting of them both remains a very distant goal indeed. This goal, however, cannot be reached without first coming to grips with concepts.

Another crucial difference between Metacat and the other approaches mentioned above (and, for that matter, the majority of approaches being pursued in cognitive modeling today) is Metacat's commitment to modeling concepts within a microdomain. Accordingly, a second bedrock assumption of Metacat (and of Metacat's predecessors) is the belief that only by "starting small" will it be possible to penetrate the deep mysteries surrounding the notion of reference and the meaning of symbols within computational systems—and, in particular, the notion of self-reference. Grappling with these issues is best done in the context of an idealized, “frictionless” world free of the confusing and obscuring clutter of complicated “real-world” domains. In this regard, work on Metacat can be viewed as following in the tradition of earlier AI projects (mostly from the 1970s) that took the idea of microdomains to heart, such as Terry Winograd’s SHRDLU program, which conversed with a human interlocutor in impressively sophisticated English about a simulated world of toy blocks [Winograd, 1972], and Anthony Davey’s Proteus program, which generated commentaries on Tic-Tac-Toe games played against a human opponent in equally impressive and sophisticated English [Davey, 1978].

As far as possible future work on Metacat is concerned, there are quite a number of directions in which research on this project could conceivably proceed. A few examples of extensions to the program that could be made are discussed below, in roughly increasing order of complexity:[1]

- A right-hand side vertical mapping could be constructed between the modified string and the answer string when the program runs in justify mode, in addition to the usual left-hand side mapping between the initial string and the target string. This would reflect more closely how people make sense of answers provided to them. For example, when people are shown the answers below,

  $$\boxed{\begin{array}{l}\boldsymbol{abc} \Rightarrow \boldsymbol{abd}\\ \boldsymbol{xyz} \Rightarrow \boldsymbol{dyz}\end{array}} \qquad \boxed{\begin{array}{l}\boldsymbol{rst} \Rightarrow \boldsymbol{rsu}\\ \boldsymbol{xyz} \Rightarrow \boldsymbol{uyz}\end{array}}$$

  they instantly notice, in the case of the first problem, the two salient ***d***’s in ***abd*** and ***dyz***, or, in the case of the second problem, the two salient ***u***’s in ***rsu*** and ***uyz***. This immediately “gives away the game” by suggesting a crosswise

[1] Some of these extensions were suggested in [Mitchell, 1990], but bear repeating here.

mapping between these strings—and, likewise, between ***abc*** and ***xyz*** (or ***rst*** and ***xyz***). In fact, the ability to make a right-hand side vertical mapping was first suggested in the original Copycat proposal [Hofstadter, 1984a], long before Copycat had been implemented. Even now, this particular idea still awaits implementation.

- Groups of arbitrary letters could be constructed on the basis of spatial proximity (*e.g.*, the group ***xem*** in the string ***aaaxemttt***, or the three ***mxb*** groups in ***mxbmxbmxb***), or on the basis of symmetry (*e.g.*, the whole-string group ***axxgggxxa***). In a similar fashion, groups could be based on more complex types of bonds between letters, such as simultaneous letter-category and group-length bonds in ***rssttt***; bonds between spatially non-adjacent letters that together form a figure/ground pattern (*e.g.*, the letters ***p***, ***q***, and ***r*** in ***pxqxrx***); or bonds based on new concepts formed from the composition of existing concepts (*e.g.*, "double successorship" bonds within ***ace***).

- More complex descriptions of objects within strings could be made, such as "the rightmost letter of the leftmost group" (*e.g.*, the rightmost ***a*** of ***aaabbbccc***), "the rightmost letters of all objects in the string" (*e.g.*, the letters ***n***, ***h***, and ***q*** in ***lmnfghopq***), "the third letter from the leftmost letter of the string" (*e.g.*, the ***c*** in ***abcdef***), "the next-to-leftmost letter" (*e.g.*, the ***j*** in ***ijklm***), or "the next-to-last letter (in the alphabet)" (*e.g.*, the ***y*** in ***wxy***).

- The information stored in answer descriptions could be expanded to include temporal information about the overall structure of a run. In other words, in addition to storing themes characterizing the essential ideas underlying an answer, answer descriptions could include information about the *pathway taken* in discovering the answer. For example, the answer description for ***wyz*** to the problem *"**abc**⇒ **abd; xyz**⇒ **?**"* might include information to the effect that "I *first* hit a snag, but *then* restructured my view and saw a far deeper similarity between ***abc*** and ***xyz***", or, for the answers ***mrrkkk*** and ***mrrjjjj*** to the problem "***abc***⇒ ***abd***; ***mrrjjj***⇒ ?", "I *first* found an answer based on little structure, but *then* noticed a pattern, explored it more deeply, and discovered a hidden layer of structure that revealed a much stronger degree of similarity between ***abc*** and ***mrrjjj***". This type of extension would make Metacat's Episodic Memory much more *episodic* than it currently is. Accordingly, extending the program in this way should be given high priority in any future work on this project.

- Another important way in which answer descriptions could be enriched would be to relax the restrictions currently imposed on the types of themes that can be included in them. As was mentioned in section 4.7.1 of Chapter 4, the current version of the program allows only themes of the category *String-Position*, *Alphabetic-Position*, *Direction*, *Group-Type*, or *Bond-Facet* to be included in answer descriptions (and only non-identity themes in the case of *Bond-Facet*). These restrictions were imposed in order to improve Metacat's ability to sensibly characterize its answers. (For example, in the problem "***abc***⇒ ***abd***; ***ijk***⇒ ?", allowing *Letter-Category* themes to be included in answer descriptions would likely mislead the program into regarding the letter-category differences between ***abc*** and ***ijk*** as being a key idea behind the answer ***ijl***.) In a way, the restrictions currently placed on Metacat's answer descriptions are akin to the restrictions that were placed on rules in Copycat, and should thus be viewed as a temporary interim solution, which should eventually be generalized.

- More cognitively plausible mechanisms for memory indexing and retrieval are needed. In the current version of the program, when a new answer is discovered, the newly-created answer description is compared with all other answer descriptions stored in memory, in order to determine the new activation levels of the stored descriptions—and hence which answers will be recalled by the program as a result of having found the new answer. This approach is adequate if only a few answer descriptions exist in memory, but very quickly breaks down if many descriptions exist. In other words, the current mechanisms for memory retrieval and reminding in Metacat do not scale up, and are thus unsatisfactory. Accordingly, for Metacat to develop into a psychologically realistic model of memory and reminding, this issue must be squarely addressed, and should thus be given high priority in any future work on the project.

- Concepts about *analogy-making in general* in the letter-string microworld could be given to the program. This would significantly increase Metacat's degree of "meta-ness", since such concepts would provide the program with a much richer conceptual vocabulary for describing and comparing analogies. For example, such concepts might include the idea of *bridge*, *mapping*, *rule*, *theme*, *slippage*, *snag*, *pressure*, *answer*, *pattern*, and the concept of *concept* itself, to name but a few.

In particular, including meta-level concepts about the process of analogy-making itself would allow Metacat to characterize entire analogy *problems* (not just individual answers to problems) in terms of "the issues that they are about", and would thereby allow the program to notice connections and distinctions between analogy problems as a whole. For example, the problem ***"abc ⇒ abd; xyz ⇒ ?"***, in its essence, is about being forced to reinterpret a situation in response to an unexpected snag, which may then lead to a kind of paradigm shift that results in the discovery of a far more elegant way of interpreting the situation. Likewise, the problem ***"eqe ⇒ qeq; abbbc ⇒ ?"*** can also be viewed in a very similar way. Giving Metacat the ability to appreciate such abstract similarities between analogy problems as a whole would be an excellent topic for future research.

Once Metacat is capable of looking at a particular problem and identifying the issues that lie at its core (*i.e.*, the ideas that motivated the invention of the problem in the first place), the next major step would be to imbue the program with the ability to take an interesting idea for a problem (perhaps supplied by a human), and proceed to *invent* one or more problems on its own that are "about" that idea. Of course, it would also be crucial to give the program itself the ability to come up with its own "interesting ideas". To do this, Metacat would need to have even-more-meta-level types of concepts, such as the notion of "decoy answer" (*i.e.*, an answer that catches the eye quickly but that has little depth) versus "elegant hidden answer" (*i.e.*, an answer whose qualities are good in many ways, but that does not jump to the eye at once). These two concepts actually are intertwined, in that one may construct a problem deliberately to have both a decoy answer and an elegant hidden answer, as in the problem *"**apc** ⇒ **abc**; **opc** ⇒ **?**"* discussed in Chapter 2 on page 50. Explicit representation of these kinds of very meta-level concepts would be needed in a program that could make up high-quality analogy problems on its own. These kinds of concepts are also deeply related to the issue of humor.

Finally, in conclusion, it is interesting to note that concurrent work by John Rehling and Douglas Hofstadter on the Letter Spirit project, described briefly in section 2.1.4 of Chapter 2, seems to be converging, in many ways, on the same set of fundamental issues at the heart of the Metacat project.

As will be recalled, Letter Spirit is concerned with creative artistic design and the perception of visual style in an idealized microworld of letterforms (called *gridletters*). Recent development of the program has involved incorporating architectural components that, in particular, share much of the flavor of Metacat's Themespace [Rehling, 1997; Rehling, 1999]. Briefly, the main components of Letter Spirit include the *Examiner*, which classifies a given letterform as one of the 26 possible lowercase letters of the alphabet; the *Adjudicator*, which uses the output of the Examiner—together with the letterform itself—both to judge how well the letterform fits into the particular letter-category assigned by the Examiner, and to determine which *stylistic* aspects of the letterform are the most salient; and the *Drafter*, which uses the stylistic information extracted by the Adjudicator to create new letterforms representing different letters of the alphabet drawn in the same style as the original letter. In turn, the new letterforms created by the program are themselves subject to examination and evaluation, in terms of letter-category quality and style, by the program itself.

The abstract stylistic information about letterforms, extracted by the Adjudicator, is explicitly represented by structures called *stylistic properties*, which are stored in Letter Spirit's *Thematic Focus*. In general, these structures exert strong top-down pressure on processing, guiding the program in its creation of new letterforms of a particular style, which may in turn cause new stylistic properties to be noticed and explicitly represented in the Thematic Focus. Stylistic properties are thus, in some sense, analogous to Metacat's themes, since they characterize gridletters (*i.e.*, the program's concrete perceptual data) at an abstract level of description, and can in turn influence the behavior of the program as it watches and responds to its own activity. Furthermore, the ongoing development of Letter Spirit has brought out, in very clear ways, the central and indispensable role played by self-watching in creativity. It will be interesting to see whether (or to what extent) future FARG work on Letter Spirit and Metacat continues to converge on a common set of fundamental ideas.

In summary, the work on Metacat described in this dissertation has attempted to address the long-term goals set forth in [Hofstadter and FARG, 1995, Chapter 7] for the further development of the Copycat project. To some extent, this effort has succeeded, although—to be sure—in a far-from-complete way. A great deal of work remains to be done. It is my hope that the work presented here will serve as another stepping stone along the path toward a deeper understanding of human cognition in all of its profound subtlety and complexity.

APPENDIX: RANDOM NUMBER SEEDS

This appendix lists the random number seeds used in creating the figures and sample runs of Metacat presented in Chapter 5. In each case, the actual expression that begins the run is given.

Sample runs (section 5.2)

Run 1: `(mcat abc abd mrrjjj mrrjjjj 1092119323)`

Run 2: `(mcat xqc xqd mrrjjj mrrkkk 1248075611)`

Run 3: `(mcat rst rsu xyz uyz 2330176791)`

Run 4: `(mcat abc abd xyz dyz 2836825623)`

Run 5: `(mcat xqc xqd mrrjjj mrrjjjj 3729474543)`

Run 6: `(mcat eqe qeq abbbc aaabccc 789090523)`

Run 7: `(mcat abc abd xyz 3852097033)`

Run 8: `(mcat eqe qeq abbbc 3557912874)`

Answer comparison and reminding (section 5.2.3)

abc/xyd: `(mcat abc abd xyz xyd 1760747975)`

abc/wyz: `(mcat abc abd xyz wyz 3100511611)`

abc/dyz: `(mcat abc abd xyz dyz 2107869027)`

rst/xyu: `(mcat rst rsu xyz xyu 939480183)`

rst/wyz: `(mcat rst rsu xyz wyz 720286361)`

rst/uyz: `(mcat rst rsu xyz uyz 2330176791)`

abc/mrrkkk: `(mcat abc abd mrrjjj mrrkkk 4211806334)`

abc/mrrjjjj: `(mcat abc abd mrrjjj mrrjjjj 1092119323)`

xqc/mrrkkk: `(mcat xqc xqd mrrjjj mrrkkk 1248075611)`

xqc/mrrjjjj: `(mcat xqc xqd mrrjjj mrrjjjj 3729474543)`

eqe/baaab: `(mcat eqe qeq abbba baaab 3635369418)`

eqe/aaabaaa: `(mcat eqe qeq abbba aaabaaa 4209674874)`

eqe/qeeeq: `(mcat eqe qeq abbbc 2302461154)`

eqe/aaabccc: `(mcat eqe qeq abbbc aaabccc 789090523)`

Implausible rules (section 5.3.1)

Figure 5.4 (top): `(mcat eeqee qeeq xxixx 698282038)`

Figure 5.4 (bottom): `(mcat eeqee qeeq xxixx 175910650)`

Figure 5.5 (top): `(mcat eeqee qeeq xxixx 698282038)`

Figure 5.5 (bottom): `(mcat eeqee qeeq xxixx 4109591222)`

Poor thematic characterizations (section 5.3.2)

Figure 5.7: `(mcat aabc aabd ijkk ijll 2351730219)`

Figure 5.8: `(mcat aabc aabd ijkk hjkk 1810079903)`

Figure 5.10: `(mcat abc abd xyz 3009318743)`

Figure 5.11: `(mcat abc abd xyz 2006188493)`

Bibliography

Boden, M. A. (1991). *The Creative Mind: Myths and Mechanisms.* Basic Books, New York.

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

Chalmers, D. J., French, R. M., and 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., and 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., and LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, 18:439–477.

Davey, A. (1978). *Discourse Production.* Edinburgh University Press, Edinburgh.

Dybvig, R. K. (1996). *The Scheme Programming Language.* Prentice-Hall, second edition.

Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980). The Hearsay II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253.

Evans, T. G. (1968). A program for the solution of a class of geometric-analogy intelligence test questions. In Minsky, M., editor, *Semantic Information Processing*, pages 271–353. MIT Press, Cambridge, MA.

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

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

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

Forbus, K. D., Gentner, D., Markman, A. B., and 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.

Fox, S. and Leake, D. B. (1994). Using introspective reasoning to guide index refinement in case-based reasoning. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 324–329. Lawrence Erlbaum Associates.

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

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

Gentner, D. (1989). Mechanisms of analogical learning. In Vosniadou, S. and Ortony, A., editors, *Similarity and Analogical Reasoning*, pages 199–241. Cambridge University Press, Cambridge.

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

Hofstadter, D. R. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, New York.

Hofstadter, D. R. (1983). The architecture of Jumbo. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Proceedings of the International Machine Learning Workshop*, pages 161–170, Urbana, IL.

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

Hofstadter, D. R. (1984b). Simple and not-so-simple analogies in the Copycat domain. Technical Report 9, Center for Research on Concepts and Cognition, Indiana University, Bloomington.

Hofstadter, D. R. (1985a). Analogies and roles in human and machine thinking. In *Metamagical Themas*, chapter 24, pages 547–603. Basic Books, New York.

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

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

Hofstadter, D. R. (1987). Introduction to the Letter Spirit project and to the idea of "gridfonts". Technical Report 17, Center for Research on Concepts and Cognition, Indiana University, Bloomington.

Hofstadter, D. R. (1992). A short compendium of me-too's and related phenomena: Mental fluidity as revealed in everyday conversation. Technical Report 57, Center for Research on Concepts and Cognition, Indiana University, Bloomington.

Hofstadter, D. R. and 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.

Hofstadter, D. R. and French, R. M. (1992). Probing the emergent behavior of Tabletop, an architecture uniting high-level perception with analogy-making. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society.* Lawrence Erlbaum Associates.

Holyoak, K., Gentner, D., and Kokinov, B., editors (1998). *Advances in Analogy Research: Integration of Theory and Data from the Cognitive, Computational, and Neural Sciences*, New Bulgarian University Series in Cognitive Science, Sofia, Bulgaria.

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

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

Koestler, A. (1964). *The Act of Creation.* Macmillan, New York.

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

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

Kokinov, B. N., Nikolov, V., and Petrov, A. (1996). Dynamics of emergent computation in DUAL. In Ramsay, A., editor, *Artificial Intelligence: Methodology, Systems, Applications*. IOS Press, Amsterdam.

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

Lange, T. E. and Wharton, C. M. (1994). REMIND: Retrieval from episodic memory by inferencing and disambiguation. In Barnden, J. A. and Holyoak, K. J., editors, *Advances in Connectionist and Neural Computation Theory, Volume 3: Analogy, Metaphor, and Reminding*, pages 29–94. Ablex, Norwood, NJ.

Law, K., Forbus, K. D., and Gentner, D. (1994). Simulating similarity-based retrieval: A comparison of ARCS and MAC/FAC. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 543–548. Lawrence Erlbaum Associates.

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

McGraw, Jr., G. E. (1995). *Letter Spirit (Part One): Emergent High-Level Perception of Letters Using Fluid Concepts.* PhD thesis, Indiana University, Bloomington, IN.

Meredith, M. J. (1986). *Seek-Whence: A Model of Pattern Perception.* PhD thesis, Indiana University, Bloomington, IN. Also available as Technical Report 214, Computer Science Department, Indiana University.

Meredith, M. J. (1991). Data modeling: A process for pattern induction. *Journal of Experimental and Theoretical Artificial Intelligence*, 3:43–68.

Mitchell, M. (1990). *Copycat: A Computer Model of High-Level Perception and Conceptual Slippage in Analogy-Making.* PhD thesis, University of Michigan, Ann Arbor, MI.

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

Mitchell, M. and Hofstadter, D. R. (1990). The emergence of understanding in a computer model of concepts and analogy-making. *Physica D*, 42:322–334.

Oehlmann, R. (1995). Meta-cognitive attention: Reasoning about strategy selection. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 66–71. Lawrence Erlbaum Associates.

Oehlmann, R., Edwards, P., and Sleeman, D. (1994). Changing the viewpoint: Re-indexing by introspective questioning. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 675–680. Lawrence Erlbaum Associates.

Pirolli, P. and Bielaczyc, K. (1989). Empirical analyses of self-explanation and transfer in learning to program. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, pages 450–457. Lawrence Erlbaum Associates.

Poincaré, H. (1921). *The Foundations of Science: Science and Hypothesis, The Value of Science, Science and Method.* The Science Press, New York.

Pólya, G. (1957). *How to Solve It.* Princeton University Press, Princeton, NJ.

Ram, A. and Cox, M. (1994). Introspective reasoning using meta-explanations for multistrategy learning. In Michalski, R. and Tecuci, G., editors, *Machine Learning: A Multistrategy Approach, 4*, pages 348–377. Morgan Kaufmann, San Francisco.

Rehling, J. (1997). Automating creative design in a visual domain. In Veale, T., editor, *Computational Models of Creative Cognition.* Proceedings of the *Mind II* workshop held at Dublin City University, Ireland, September 1997.

Rehling, J. (1999). *Letter Spirit (Part Two): Automating Creative Design in a Visual Domain.* PhD thesis, Indiana University, Bloomington. (Forthcoming).

Riesbeck, C. K. and Schank, R. C. (1989). *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Sandoval, W. A., Trafton, J. G., and Reiser, B. J. (1995). The effects of self-explanation on studying examples and solving problems. In *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society*, pages 253–258. Lawrence Erlbaum Associates.

Schank, R. and Abelson, R. (1977). *Scripts, Plans, Goals, and Understanding.* Lawrence Erlbaum Associates, Hillsdale, NJ.

Schank, R. C. (1975). *Conceptual Information Processing.* North Holland/American Elsevier, Amsterdam.

Schank, R. C. (1982). *Dynamic Memory: A Theory of Learning in Computers and People*. Cambridge University Press, Cambridge, England.

Shepard, R. N. (1962). The analysis of proximities: Multidimensional scaling with an unknown distance function. *Psychometrika*, 27:125–140.

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

Thagard, P. (1989). Explanatory coherence. *Behavioral and Brain Sciences*, 12(3):435–467.

Thagard, P., Holyoak, K., Nelson, G., and 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.

VanLehn, K., Jones, R., and Chi, M. (1992). A model of the self-explanation effect. *The Journal of the Learning Sciences*, 2(1):1–59.

VanLehn, K. and Jones, R. M. (1993). What mediates the self-explanation effect? knowledge gaps, schemas, or analogies? In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pages 1034–1039. Lawrence Erlbaum Associates.

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

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

Weizenbaum, J. (1976). *Computer Power and Human Reason: From Judgment to Calculation*. Freeman, San Francisco.

Winograd, T. A. (1972). *Understanding Natural Language*. Academic Press, New York.

Zuckerman, J. (1992a). *SchemeSGL: A Symbolic Graphics Language for Chez Scheme*. Motorola, Inc., second edition.

Zuckerman, J. (1992b). *The SchemeXM Manual*. Motorola, Inc., second edition.