CHAPTER

Eighteen

Programming by Analogous Examples

ALEXANDER REPENNING AND CORRINA PERRONE

Agent Sheets, Inc., Center of LifeLong Learning & Design, University of Colorado, Boulder

S
R
L

Abtract

Analogies are powerful cognitive mechanisms that people use to construct new knowledge from knowledge already acquired and understood. When analogies are used with programming by example (PBE), the result is a new end-user programming paradigm combining the elegance of PBE to create programs with the power of analogies to reuse programs. The combination of PBE with analogies is called Programming by Analogous Examples (PBAE).

18.1 Introduction

Why do end users need to program? In a world with an ever-increasing flood of information, people become overwhelmed trying to cope with it. With the ubiquity of computer networks, the information challenge is no longer about accessing information but processing it. The direct manipulation paradigm, popularized in the 1980s, begins to break down when it is no longer feasible to directly manipulate the sources of information such as the location of *all* the files on your hard disk or *all* the emails in your in box.

End-user programming (Nardi 1993) is becoming a crucial instrument in the daily information-processing struggle. End-user programming is a form of programming done by the end user to customize information processing. Most computer end users do not have the background, motivation, or time to use traditional programming approaches, nor do they typically have the means to hire professional programmers to create their programs. Simple forms of end-user programming include the use of email filters to clean up email by directing specified emails into separate folders or the use of spreadsheets to explore the total cost of a new house.

Programming by example (PBE) is a powerful end-user programming paradigm enabling computer users without formal training in programming to create sophisticated programs. PBE environments create programs for end users by observing and recording as users manipulate information on a GUI level. For instance, in Microsoft Word PBE is used to build macros.

The focus of this chapter is the problem of PBE *reuse*. A user may find the PBE-generated program useful but may need either to generalize it or to modify it for a related yet different task. Program reuse is a well-known software design problem (Lange 1989). Reuse problems hamper productivity of a wide range of software projects, from those involving individual end-user

programmers to large distributed teams of software developers (Roschelle et al. 1999). In the PBE context, reuse poses even more complex issues. While initially PBE shields end users from having to deal with programming issues, reuse will force them to leap cognitively between two levels of representations:

- *The GUI level:* This is the level of representation featuring windows, icons, and menus familiar to users. For instance, in the case of a word processor application such as Microsoft Word, this is the level that represents content directly manipulated by users with operations such as typing in new text, formatting text, using cursor keys to navigate through a document, and so forth.
- *The program level:* This is the level of representation that captures user manipulations into programs so that they can be replayed by the users. In the case of Word, this is the level of Visual Basic. Manipulations by users are recorded as Visual Basic scripts. Users assign scripts to keyboard commands or to user-defined toolbar commands.

The *PBE representation chasm* describes how difficult it is for users to comprehend the mapping between the GUI and the programming levels. In reuse this chasm is especially problematic since users, to adapt representations at the programming level to their needs, will be required to have at least a minimal understanding of the programming level representations. For the end user with no previous exposure to programming in general or to programming in Visual Basic, this transition may be too complex and result in frustration. As a consequence, the user may just give up on the idea of end-user programming and continue to solve the original problem manually again and again for years.

Analogies are powerful cognitive mechanisms that people use to construct new knowledge from knowledge already acquired and understood. When analogies are used with PBE, the result is a new end-user programming paradigm combining the elegance of PBE to *create* programs with the power of analogies to *reuse* programs. The combination of PBE with analogies is called *programming by analogous examples* (PBAE). Analogies are an effective representation level bridging the GUI level with the program level.

In this chapter, we portray the reuse problem with two detailed examples. In the first example a Microsoft Word macro to do repetitive reformatting is created and reusability of macros is explored. This example is chosen not because the Word macro-recording mechanism is the most sophisticated PBE system but because Word has a large user base and,

consequently, readers may best be able to relate to use/reuse issues in this context. In the second example, a SimCity-like simulation is built using the AgentSheets (Repenning and Sumner 1995) simulation authoring tool. AgentSheets provides an end-user programming approach that is considerably more accessible to most users than Visual Basic. We will show that even if the program level is more accessible, there is still a need for analogies as a PBE reuse mechanism. Throughout these examples PBAE is contrasted with existing reuse mechanisms known from object-oriented programming, such as inheritance.

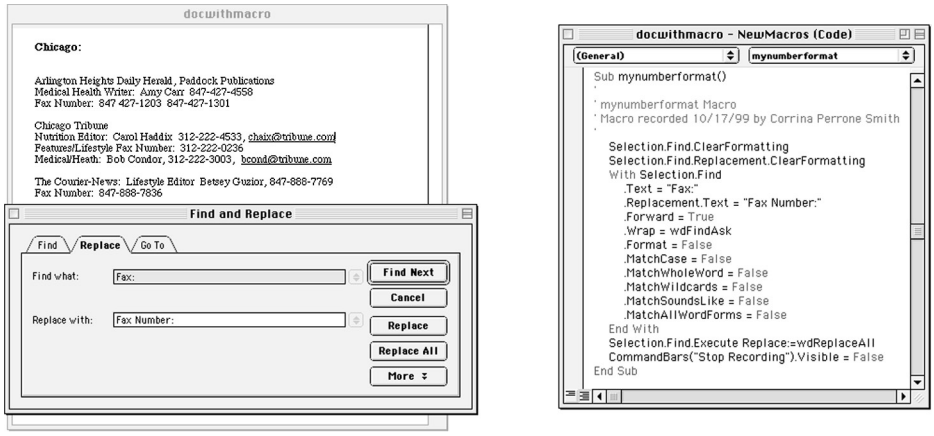
18.2 The GUI to Program Chasm

As PBE techniques become more widely adopted in both commercial and research systems, users incrementally construct and edit new behavior or commands as needed. As in traditional programming approaches, effective reuse of this new behavior is limited by underlying representations. In fact, these representations run the risk of presenting end-user programmers with the same copy/paste/modify, or inherit, reuse problem that professional programmers have struggled with for years. What makes reuse even more problematic in a PBE situation is that the programs that need to be modified for reuse have been machine generated. Typically these programs contain little information regarding how they could be changed. In this section, we describe a reuse scenario in which an end user is trying to reformat an address list automatically.

Microsoft Word and other applications allow for modular modification through the *macro* mechanism. Users are given an interface to program macros by example through the Record New Macro menu command. Additional support is given by the macro-editing toolbar and menu functions. Once a macro is named and assigned to a toolbar icon or a keyboard command, the macro is created by simply doing the task desired—in our example, repetitive reformatting. The system records keystrokes and compiles the command into Visual Basic, and the command can then be invoked by anyone editing a document.

Often a user would like to modify, and hence reuse, a macro with additional, related actions and have the entire new set of changes assigned to the same keyboard command. There are mechanisms to rename the macro, and thus copy the same behavior, but any modification to an existing macro must be done through the Visual Basic programming language. For instance, how would a user have to modify the macro to reformat not just the

FIGURE **18.1**



Creating a PBE to format all occurences of the "Fax:" into "Fax Number": the GUI level (left) shows a document edited with find-and-replace dialogue box, and the program level (right) shows a Visual Basic editor.

"Fax:" but also an "Address:" field? To modify this macro through the record function, a user would have to overwrite the old macro, go through the exact actions that were previously recorded, and then record the desired additional behavior.

To be able to reuse existing macros and make them useful in more general situations, users will have to drop from the GUI level (Fig. 18.1, left) to the program level (Fig. 18.1, right). According to the Word documentation:[1]

> Recorded macros are great when you want to perform exactly the same task every time you run the macro. But what if you want to automate a task in which actions vary with the situation, or depend on user input [. . .]? To create powerful automations, you should learn to program in Visual Basic for Applications.

The cognitive chasm between the GUI and program levels is quite large. The GUI level provides only very limited options for reuse. Even though advanced reuse *requires* making the transition into the program level, no

1. Microsoft Corporation, Getting Results with Microsoft Word 98 Macintosh Edition (Redmond, Wash.: Microsoft Corporation, 1987-1998), p.fl227.

S
R
L

intermediate steps are provided to take the user *from* the simple recording mechanism *to* the Visual Basic programming language. At the GUI level, users are limited in reuse to copy macros as black boxes into other documents or to make macros generally available to all documents. At the program level, code may be copied, pasted, and modified with the same difficulty as any other object-oriented program. At this point, the user has jumped from the comfortable environment of direct manipulation at the GUI level directly into the uncomfortable or downright frustrating programming level.

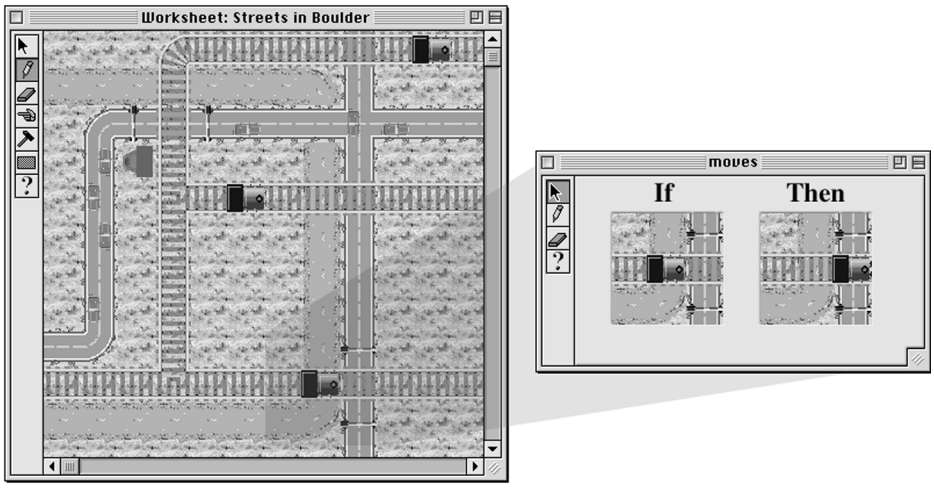
18.3 Programming by Analogous Examples

We use the AgentSheets simulation authoring tool to explain how analogies bridge the chasm between the GUI level and the program level in PBE. AgentSheets is employed to build SimCity-like simulations and export them as interactive Java applets. A large number of simulations have been built by a wide range of users, including elementary school kids and NASA scientists. A detailed description of AgentSheets can be found elsewhere (www.agentsheets.com/userforum-publications.html).

AgentSheets combines PBE with graphical rewrite rules (Cypher and Smith 1995; Lieberman 1987; Repenning 1993, 1995) into an end-user programming paradigm. Graphical rewrite rules (GRRs), which are also used in systems such as Cocoa/KidSim, and Creator, are powerful languages to express the concept of change in a visual representation. These rules declaratively describe spatial transformations with a sequence of two or more dimensional *situations* containing *objects* (Fig. 18.2, right). Situations can be interpreted with respect to objects contained and spatial relationships holding between these objects. The differences between situations imply one or more *actions* capable of transforming one situation into another. In AgentSheets, any number of GRRs can be aggregated to create complex behaviors for agents, and these agents and their behaviors combine and interact to simulate anything from heat diffusion to urban planning. We have found that it is highly likely that behavior created in one simulation will be desirable for users building other simulations.

Compared to the previously described GUI/program level chasm in Word, the chasm between the GUI and program level in AgentSheets is less pronounced. The GRR representation at the program level matches the representation at the GUI level closely. For instance, the user-produced train and train track icons found at the GUI level are also found at the program level. This representational mapping helps users comprehend programs.

FIGURE **18.2**



Programming a train to follow a train track by example in AgentSheets. When a user moves a train on a train track at the GUI level (left), AgentSheets records the movement including context and represents it at the program level as graphical rewrite rule (right).

We argue that despite this closer match between GUI and programming, it is still necessary to have a mechanism to deal with reuse.

18.3.1 Making Cars Move Like Trains: An Analogy

In the application called Sustainopolis (Fig. 18.2) used to explore public transportation issues, an end user—in this case, an urban planner—wishes to incorporate cars and streets. Noticing that Trains and Tracks are already successfully programmed, and realizing that cars move on streets similar to the way trains move on tracks, the user wishes to reuse the *move* behavior already written for the Train object and attach it to the Car object.

One approach would be for the user to start from scratch and simply demonstrate all the examples of Car and Street interactions. Unfortunately, the set of rules attached to the Train is quite large. The problem is that even for a relatively simple behavior such as making the Train follow the Track, rules had to be demonstrated to specify all of the meaningful combinations of interactions between Trains and Tracks (see also Fig. 18.5, later). Trains

S
R
L

are capable of heading in four different directions (north, south, east, and west). A complete specification accounting for fifteen different train track pieces (straight tracks, vertical and horizontal; four orientation of curves, four different T intersections, one crossing, and four orientations of cul de sacs) requires 256 rules. These rules specify behavior in situations such as "If the train drives into a cul-de-sac, which way should the train go if it has a choice of going left or right?" If the Tracks are carefully arranged in the simulation, a good number of rules can be eliminated, but the point remains that it must have taken quite a while and some patience to demonstrate the complete set of rules. Given that in our scenario this behavior already exists, it would be preferable to find a way to avoid having to demonstrate the entire set of rules again to specify the conceptually similar interaction between Cars and Streets.

A different approach is to copy all the Train's rules into the Car and manually edit them by substituting corresponding icons. In most GRR systems, the user would then copy all the rules from Trains and Tracks to Streets and Cars and then simply substitute Cars for Trains and Streets for Tracks. But is time saved? This is basically the same arduous, error-prone process a programmer must go through to reuse a program or a Word user must go through to reuse a macro. Without the understanding of what the user intends, can we facilitate the reuse process? We argue that the mechanism that allows an end user to program allows reuse as well.

Programming by analogous examples supports the reuse of previously recorded example programs. Instead of creating the behavior of cars from scratch either through demonstration or through manual editing, users generate a complete set of rules by specifying an analogy.

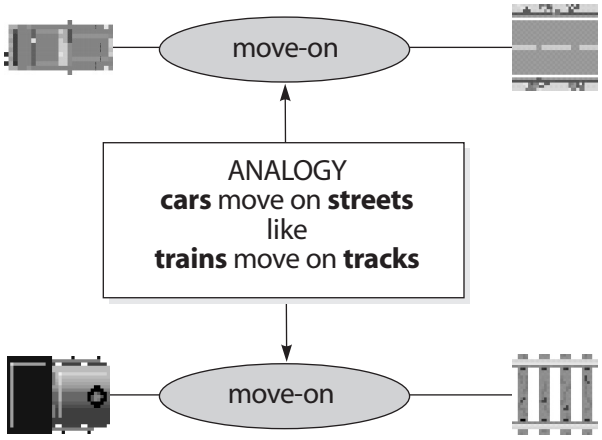
It is the *relationship* between Trains and Tracks that the user wishes to reuse by applying it to Cars and Streets (Fig. 18.3). In contrast to object-oriented programming, in which inheritance is used to define an ontology of object classes, analogies are used to define an ontology of object relationships represented by verbs such as "move."

The Analogy dialogue box (Fig. 18.4) is used to define analogies. The result is that the behavior programmed via graphical rewrite rule for Trains moving on Tracks is transferred to Cars on Streets. The verb "moves" is differentiated for Trains and Cars, so that Trains do not now move on Streets, nor will Cars move on Tracks, because no relationship is implied between Trains and Streets or Cars and Tracks.

Here, reuse is expressed in terms of existing objects, and no abstractions are required. The result of the analogy is a new set of GRRs attached to Cars that allow Cars to move on Streets. All the rules generated by analogy are

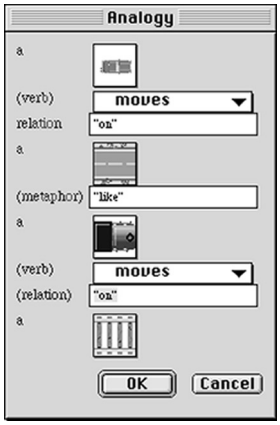
S
R
L

FIGURE **18.3**



Transferring the relationship between objects.

FIGURE **18.4**



Making an analogy.

like ordinary graphical rewrite rules in that the user can edit them to deal with exceptions to the analogy.

In summary, PBE systems often feature representations that, through the use of analogies, serve as bridge between the GUI level and the program level. Here, a user formulating an analogy created new behavior out of existing behavior. This style of reuse-enhanced programming in which an initial set of programs is created through PBE and then extended through analogies is what we call programming by analogous examples.

18.4 Discussion

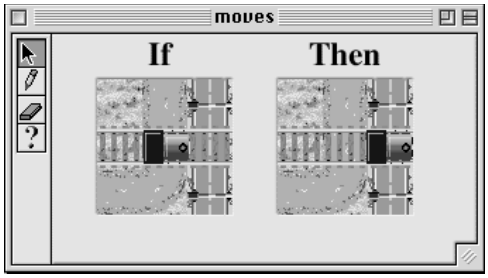
So far we have described the user perspective of PBAE. What is the depth at which an analogy mechanism needs to "understand" a program to create an analogous program? On the one hand, mere syntactic symbol substitution will not be sufficient for the transformation of nontrivial programs. On the other hand, a deep understanding of the program semantic is not only an extraordinarily hard and yet unsolved artificial intelligence (AI) problem but would probably also require users to annotate programs extensively with semantic information to enable analogies. We discuss next how PBAE moves beyond simple substitution without becoming an AI complete problem. We also list some of the problems when trying to use inheritance found in object-oriented programming as an improvement over simple substitution.

18.4.1 Beyond Syntactic Rewrite Rules

A first step toward creating more usable and reusable rewrite rules it to move from *syntactic* rewrite rules to *semantic* rewrite rules. To transform programs, it is important for programs to include semantic meta-information. This meta information can capture—to a limited degree—what the programs is doing and how it is doing it. First-generation rewrite rule–based systems such as AgentSheets91 and later Cocoa, Vampire, and Creator operate only on a syntactic level. A rule such as the one shown in Figure 18.5 is rich in meaning to a human being but cannot be interpreted by syntactic rewrite rule systems.

The system does not know what trains and train tracks are, nor does it have any sense of the functional relationship between a train and the train tracks. To a syntactic rewrite rule, systems objects are uninterpretable

FIGURE **18.5**



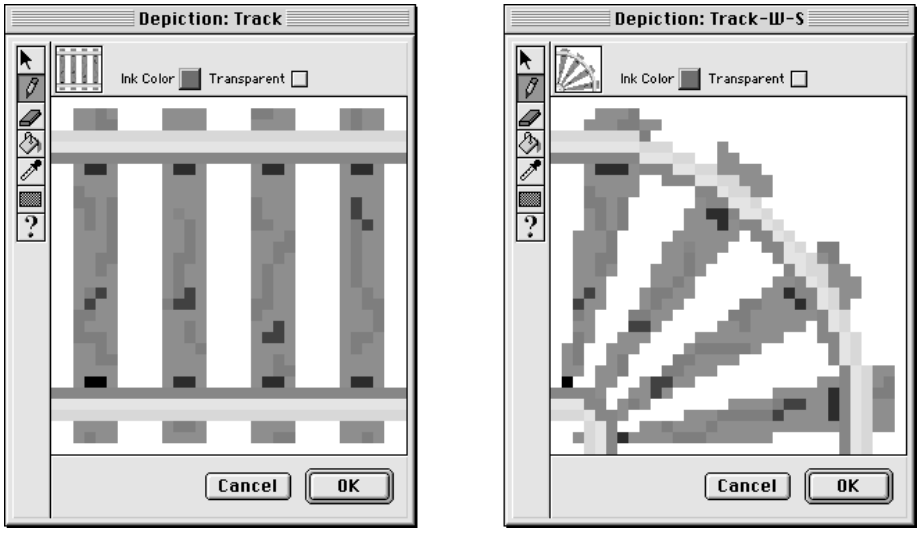
A rule that makes a train follow a train track.

bitmaps, and scenes are patterns containing objects. Without semantics, the system cannot allow users to generalize or reuse behaviors such as the behavior of a train following a train track. The *meaning* of a syntactic rewrite rule remains strictly in the programmer's head.

The lack of semantics not only makes reuse difficult but also creates a significant problem for building new behaviors from scratch. Suppose a user would like to create a complete set of rules that makes trains follow train tracks (see Figure 18.6). Through the power of programming by example, the rule described earlier is defined in no time. With this one rule, trains can follow any number of horizontal train tracks. However, a track system is not limited to horizontal track pieces—there are also curves, intersections, and so forth. Fortunately, the user does not have to draw all these different track pieces manually but can have AgentSheets generate new bitmaps automatically by transforming the original depiction drawn by the user. The transformation of bitmaps is a syntactic transformation.

Unfortunately, the complete behavior of trains following tracks is made a bit more complex since we now have fifteen different track pieces (Fig. 18.8 on page **XXX**). A very large set of rules would have to be defined to specify all the combinations of track pieces that the train is currently on, the track pieces the train can move to, and the four different directions that the train can move onto. What started as a very simple single rule quickly has turned into a tedious PBE exercise. The lack of semantics dramatically reduces the scalability of a PBE approach. In effect, users get trapped by affordances of the programming environment. They are off to quick start to create a simple behavior only to be "trapped" by the programming approach later when trying to create more complex behaviors.

FIGURE **18.6**



Syntactic transformation: AgentSheets has created a curve by "bending" a track icon designed by the user.

To scale the combination of PBE with graphical rewrite rules, it is necessary to add at least a limited amount of semantics. Semantics simplifies at least two kinds of programming:

- *Original programming: Building new behavior:* Semantic graphical rewrite rules simplify programming by allowing end users to annotate the objects that are programmed with semantic information. For instance, AgentSheets94 (Repenning 1994a&b) allows users to annotate objects with connectivity information describing whether an object has input and/or output ports in a certain direction. AgentSheets can transform agents syntactically (how an agent looks) as well as semantically (what an agent means). The same kind of annotation makes sense for all kind of agents representing agents in a context of flow. Examples include roads, train tracks, rivers, and wires.
- *Programming through reuse: Analogies:* The same semantics that help a user create original programs simply can also support reuse. The

S
R
L

following section will describe how this same connectivity semantics enables reuse through analogies.

18.4.2 From Substitutions to Analogies

The analogy menu specified in Figure 18.4 only uses four icons out of a much larger set of icons representing Cars and Trains. There are Cars and Trains with different headings as well as Streets and Tracks representing different topologies. For a Train to move properly on all kinds of Tracks, all of these icons are available to AgentSheets. A mechanism that would merely substitute the Car and Street icons for the Train and Track icons to create a set of analogous rewrite rules would not be very useful because it would only match a very small subset of all relevant rules. How does the analogy system establish the more general mapping between all these different but related items?

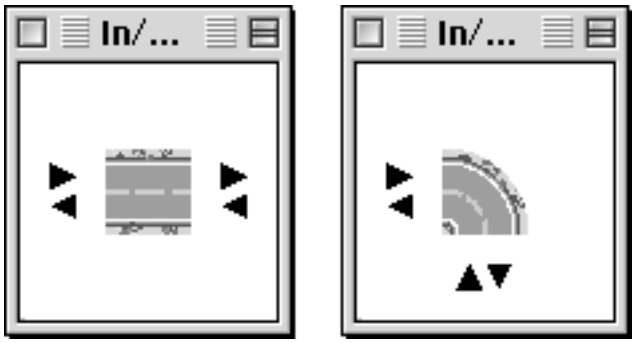
Analogy mechanisms need to have access to some minimal semantic information establishing more general correspondence (Chee 1993; Goldstone, Medin, and Gentner 1991; Medin, Goldstone, and Gentner 1993). It is no trivial matter for a computer system to substitute correctly *at all the necessary levels* required to render the resulting code useful and usable without further editing by a human. Lewis (1988) proposes a concept called *pupstitution* to decrease the brittleness inherent in straight copy and paste techniques.

In the AgentSheets substrate, systematicity in analogy (Perrone and Repenning 1998) is facilitated by connectivity semantics. Structural and behavioral similarity between the base and target objects defines systematicity. Analogies are made by specifying which relationship should be transferred. Cars and Trains are objects that can be as simple or complex as a user desires—they may have a single applicable behavior or many. However, when the user specifies that Cars move like Trains, it is this specific behavior that is transferred. For this to occur with a high degree of systematicity, and therefore effectiveness to the user, the Street and Track objects must be structurally similar.

AgentSheets assures this in two ways. Structure is created by the base icons created for the gallery. Syntactic transformations are applied by the system at the user's request that will automatically create meaningful variations to illustrate intersections and curves, as well as directional orientation (Repenning 1995). This creates the visual variations necessary to place the Agents on the AgentSheets worksheet and saves the user many hours of

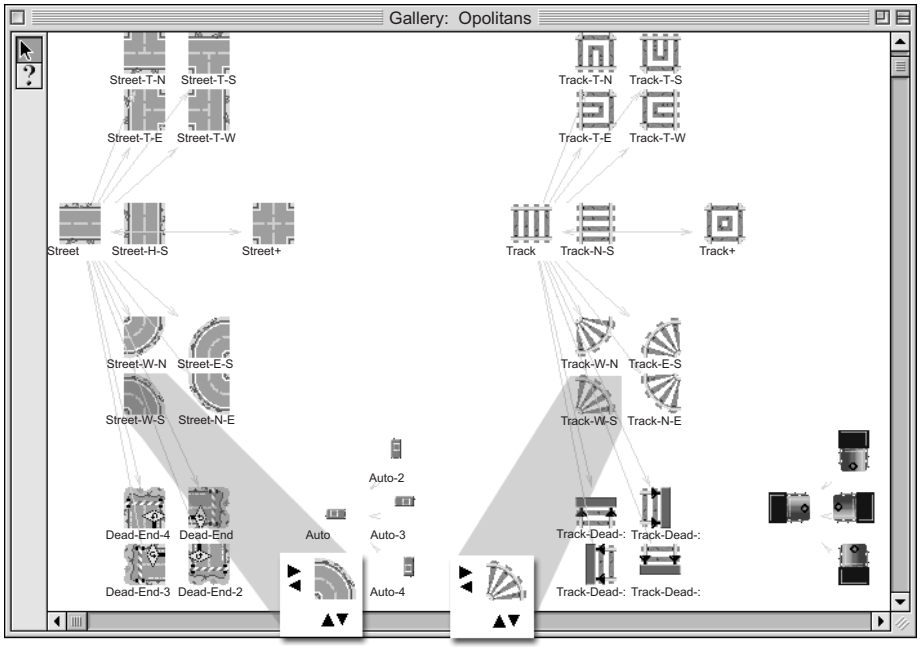
S
R
L

FIGURE **18.7**



Agent depictions are also transformed semantically. Semantics capture input/output information representing the implied connectivity of a depiction.

FIGURE **18.8**



Icons are transformed syntactically and semantically. Users only define the basic root icons, such as the horizontal Street segment, which then get transformed into icons such as curves in terms of appearance and meaning. The semantic information is used to match up icons for analogies.

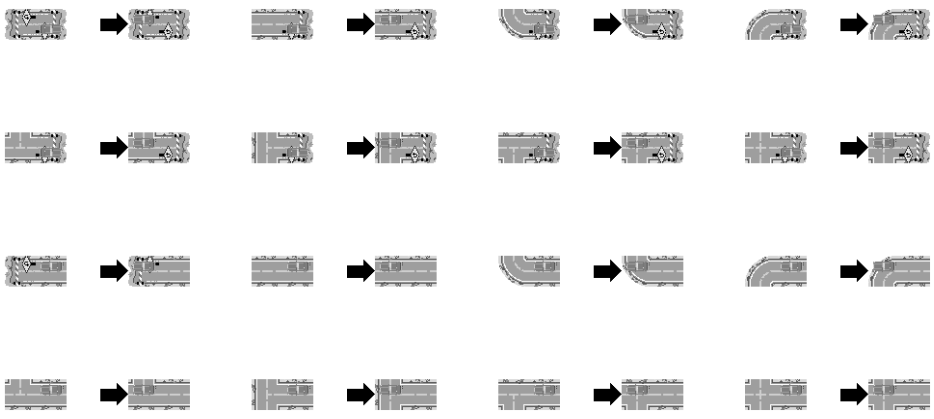
S
R
L

work upon the icon depictions. Icons are then annotated by the user with semantic information such as connectivity (Figure 18.7), which is potential behavior. Connectivity defines input and output ports of each icon. For instance, a horizontal Street segment icon has inputs/outputs to its left and right. Transforming the annotated Track creates an entire family of Track icons (Figure 18.8, right), which essentially act as throughputs for objects that move on them.

This semantic and syntactic connectivity information about an icon is used to support pupstitution in PBAE. By specifying the connectivity of an icon (as in Figure 18.8, left) and then transforming it, the *move* GRR now acquires a useful dimension of complexity. After the analogy is made, Cars have a rule set isomorphic to the Train rule set, that is, Cars know how to move on all kinds of Streets (see Figure 18.9). The system uses the semantic information available to ensure a high systematicity between the base and target; thus, the correct mapping between Tracks and Streets is made without the user's intervention.

The general insight is that a little semantics is necessary to enable meaningful analogies. On the one hand, this means that users will have to provide some additional up-front information to annotate their designs with minimalist semantics. On the other hand, this kind of semantic annotation dramatically improves the reusability of behavior. In the case of

FIGURE **18.9**



Only the first few out of a set of 256 rules created by AgentSheets' "Semantic rewrite rule" system to define the "Cars follow Streets *like Trains follow Tracks" behavior.*

S
R
L

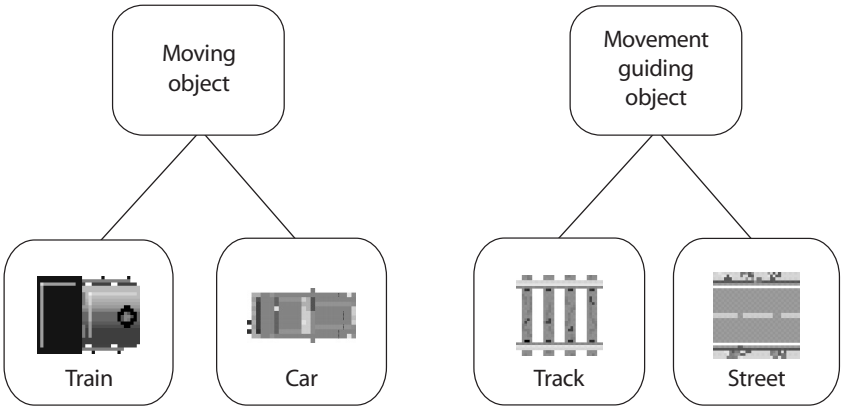
AgentSheets, users only need to provide semantics information to the base agents. For instance, they need to specify the connectivity of a Street agent that they have defined. AgentSheets can automatically transform agents representing flow conductors such as streets, wires, or pipes syntactically (by applying geometric transformations to the agent bitmap) as well as semantically (by applying topological transformations to the agent's connectivity).

18.4.3 Reuse through Inheritance

In object-oriented programming, inheritance can serve the role of reuse mechanism. An object subclass not only inherits the behavior of the superclass but can even overwrite and extend this behavior. How would inheritance have worked for our trains and cars example? One easy way to get analogous behavior is to make Car a type of, or subclass of, Train and Street a subclass of Track. While this approach will produce the desired behavior because of inheritance, it is ontologically unsound, and changing the object hierarchy in this way produces a misleading model based on weak design.

To correct this problem, it is expected that the end-user become a bit more of a programmer. In the class hierarchy, Cars become siblings of Trains and Streets sibling of Tracks by creating two abstract superclasses: Moving Object and Movement Guiding Object (Figure 18.10).

FIGURE **18.10**



S

Reusing through inheritance.

To enable both Trains and *Cars* to move, a generalized graphical rewrite rule needs to be expressed in terms of Moving Object and Movement Guiding Object. While this new class hierarchy may be ontologically sound, it introduces serious problems:

- *Need for abstractions:* Abstractions are nontrivial for end users to make and are hard to represent visually. This is especially true in the case where objects are represented by user-defined icons. How would abstract objects embodying Moving Objects and Movement Guiding Objects look and who would have to draw them?
- ***Overgeneralization:*** If city traffic is run with this new representation in place, Trains can now move on Streets and Cars will now move on Tracks, which, although theoretically possible, should not be allowed to happen in the urban planning domain application. To get around this real-life constraint, the behavior of Trains and Cars would need to be specialized again to prevent these unwanted combinations of Moving Objects and Movement Guiding Objects.

Inheritance is a powerful means of generalization that could increase the usefulness of PBE. However, the tension between inheritance and PBE with respect to representational concreteness is hard to resolve. Inheritance is pulling toward the need to introduce and manipulate abstract representations, whereas PBE is pulling toward the need to provide highly concrete representations that can be manipulated by end users.

18.5 Conclusion

In PBE, the representation that the user sees at the GUI level may indeed be vastly different from the representation a user is faced with at the program level. In PBE systems, special representations are often used to make the transition between the two levels easier for an end user. These representational features enable programming by analogous examples, which in turn simplifies program reuse.

Our work with PBAE is at an early but promising stage. The combination of *some* semantic information with structural information has allowed reuse of complex behaviors in the context of interactive simulations. Of course, the more similarity between behavior the user has created and wants to reuse and the target behavior makes for a better analogy and

minimal editing. We are continuing work on PBAE to allow for greater differentiation by a user trying to determine appropriate analogical matches. For instance, Cars moving on Streets and Electricity moving through Wires also share similarity, but Electricity can go both directions at an intersection. Therefore, an analogy made between the two creates accident-prone Cars. Next steps will also explore the use of PBAE in nonsimulation application domains.

Acknowledgements

The authors wish to acknowledge Clayton Lewis, Braden Craig, and the members of the Center of LifeLong Learning & Design at the University of Colorado. The research was supported by the National Science Foundation under grants No. DMI-9761360, RED 925–3425, and Supplement to RED 925–3425. AgentSheets research and AgentSheets Inc. are supported by the National Science Foundation (REC 9804930, REC-9631396, and CDA-940860).

References

Chee, Y. S. 1993. Applying Gentner's theory of analogy to the teaching of computer programming. *International Journal of Man Machine Studies,* 38: 347–368.

Cypher, A., and D. C. Smith. 1995. KidSim: End user programming of simulations. In *Proceedings of the 1995 Conference of Human Factors in Computing Systems* (Denver, Colo.). ACM Press.

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

Lange, B. M. 1989. Some strategies of reuse in an object-oriented programming environment. In *CHI '89* (Houston, Tex.). Association for Computing Machinery.

Lewis, C. 1988. Some learnability results for analogical generalization. Technical Report No. CU-CS-384–88, University of Colorado, Computer Science Department.

Lieberman, H. 1987. An example-based environment for beginning programmers. In *Artificial Intelligence and Education* ed. R. W. Lawler and M. Yazdani. Norwood, N.J.: Ablex.

Medin, D. L., R. L. Goldstone, and D. Gentner. 1993. Respects for similarity. *Psychological Review 100,* no. 2: 254–278.

Nardi, B. 1993. *A small matter of programming.* Cambridge, Mass.: MIT Press.

Perrone, C., and A. Repenning. 1998. Graphical rewrite rule analogies: Avoiding the inherit or copy & paste reuse dilemma. In *Proceedings of the 1998 IEEE Symposium of Visual Languages* (Nova Scotia, Canada). Computer Society.

Repenning, A. 1993. Agentsheets: A tool for building domain-oriented visual programming environments. In *INTERCHI '93: Conference on Human Factors in Computing Systems* (Amsterdam). ACM Press.

———. 1994a. Bending icons: syntactic and semantic transformation of icons. In *Proceedings of the 1994 IEEE Symposium on Visual Languages* (St. Louis, Mo.). IEEE Computer.

———. 1994b. Programming substrates to create interactive learning environments. *Journal of Interactive Learning Environments 4,* no. 1 (*Special Issue on End-User Environments*) 45–74.

———. 1995. Bending the rules: Steps toward semantically enriched graphical rewrite rules. In *Proceedings of Visual Languages* (Darmstadt, Germany). IEEE Computer Society.

Repenning, A., and T. Sumner. 1995. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer 28,* no. 3: 17–25.

Roschelle, J., C. DiGiano, M. Koutlis, A. Repenning, J. Phillips, N. Jackiw, and D. Suthers. 1999. Developing educational software components. *IEEE Computer 32,* no. 9: 50–58.

S
R
L

S
R
L