Repenning, A., "Bending the Rules: Steps toward Semantically enriched Graphical Rewrite Rules," *Proceeding of Visual Languages*, Darmstadt, Germany, 1995, pp. 226-233.

Bending the Rules: Steps Toward Semantically Enriched Graphical Rewrite Rules

Alexander Repenning

Department of Computer Science
Center for LifeLong Learning and Design
University of Colorado, Boulder CO 80309-0430
Phone: (303) 492-1349, Email: ralex@cs.colorado.edu
http://www.cs.colorado.edu/~ralex/

Abstract

Graphical rewrite rules, as a form of end-user programming, suffer from their implicit underlying model. Interpretation of rewrite rules limited to syntactic properties makes it laborious for end users to define non-trivial behavior. Semantically enriched graphical rewrite rules have increased expressiveness, resulting in a significantly reduced number of rewrite rules. This reduction is essential in order to keep rewrite rule-based programming approaches feasible for end-user programming. The extension of the rewrite rule model with semantics not only benefits the definition of behavior but additionally it supports the entire visual programming process. Specifically the benefits include support for defining object look, laying out scenes consisting of dependent objects, defining behavior with a reduced number of rewrite rules, and reusing existing behaviors via rewrite rule analogies. These benefits are described in the context of the Agentsheets programming substrate.

1. Introduction: The Economy of Semantics

The visual character of graphical rewrite rules combined with the increased ability of computers to represent pictures and to support the manipulation of pictures makes them an appealing choice as an end-user programming mechanism [8]. Graphical rewrite rules can be used in application domains including simulations and animations. In order to be an effective end user programming approach, environments based on rewrite rules must be expressive [3]. That is, they should allow the *definition* and *reuse* of non-trivial behavior typical for an anticipated set of problems with reasonable programming effort. Furthermore, programming environments that allow end users to create their own objects should also support the definition of object look and provide mechanisms to link look with behavior.

Graphical rewrite rules declaratively describe spatial transformations with a sequence of two or more dimensional *situations* containing *objects* (Figure 1). 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. Figure 1 depicts a rewrite rule containing cars, traffic lights, and roads. One possible interpretation for the action implied is that the car moved.

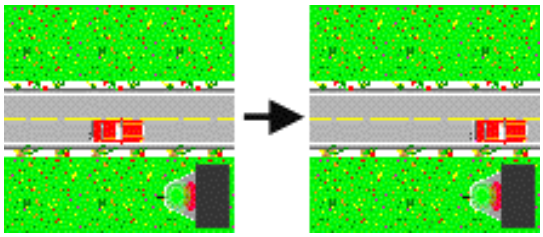

Figure 1. Rewrite Rule = <Situation$_1$, Situation$_2$> Action(s)

The *implicit* nature of graphical rewrite rules, due to the lack of sufficient interpretational clues, makes it hard for end users to assess their general applicability. Would the rule shown also work for other cars, other roads? To what other related situations would the rule apply? Is the car in situation 1 the same car as in situation 2? Unless rewrite rules serve as literal substitutions of one situation with another it is important for the end user to understand *how rules are interpreted* in order to allow for generalization of objects and spatial relationships between objects.

Kirsch [5],who devised an early version of rewrite rules, recognized the problems due to the implicit nature of rewrite rules and commented on their scalability to more complex problems: “It is not clear, however, how to extend the implicit underlying model used here to other pictorial sources of greater interest and importance.” Indeed, this problem largely remains with today’s graphical rewrite rule- based programming environments limiting their usefulness.

More realistically depicted objects in rewrite rules lead to higher interpretational expectations of humans with respect to what the meaning of individual objects is and in what

kinds of spatial relationships objects are involved in. Simple objects are not suggestive for complex interpretations. Kirsch used abstract character symbols such as V, H, and L in his rules. In BitPict [4], objects are simple black or white pixels. However, in rewrite rules featured in ChemTrains [2], Vampire [7], Agentsheets [10], and KidSim [15] objects can be much more complex and realistic looking. The rule depicted in Figure 1 is likely to be interpreted as a situation in which a car on a road is *facing* a traffic light. In situation 2 the car seems to have moved *forward*. We begin to realize the tension between the interpretation of the situations by the machine and the human. The understanding of the machine is extremely limited and purely *syntactical*. The machine has no way to identify the objects in the situations as cars, roads, and traffic lights. Because of this lack of semantic information, the notion of the car facing the traffic light cannot be internalized by a machine because it does not understand the notion of the car being a moving thing with a direction. Without this sense of direction the concept of moving forward is understood in a restrained Euclidean sense as moving to the right. With this interpretation of the above rule the car would not know how to follow a vertically oriented road.

The key is to provide mechanisms to users that allow them to add semantics to objects in order to overcome the implicit nature of rewrite rules. In this paper we claim that economical representations of semantics can be added to a system by end users in ways that lead to a variety of benefits for the definition of behavior as well as the look of objects. Specifically, adding semantics to rewrite rule environments simplifies:

1) ***Defining Object Look***. Quite often there is a strong link between the look and the behavior of objects. For limited domains tools can be built that help to generate the look of objects based on their semantics.

2) ***Laying Out Dependent Objects***. Object semantics that have consequences for how groups of related objects are created can be used to automatically lay out objects. This saves time and increases consistency.

3) ***Defining Object Behavior***. Semantics can be used by graphical rewrite environments to interpret rules in more general ways. Expressiveness increases because large number of syntactical rewrite rules can be replaced with a significantly smaller number of semantically enriched rewrite rules.

4) ***Reusing Behavior***. Semantic information capturing behavioral relationships between objects can be reused via analogies and mapped onto a different set of objects.

The following sections illustrate how *connectivity*, as a form of semantic information, supports the drawing and programming process. Then the implications of using connectivity with respect to 1) - 4) are discussed in the context of the Agentsheets environment. Finally, some alternative solution approaches are contrasted with semantically enriched rewrite rules and some shortcomings of semantic rewrite rules are outlined.

2. Connectivity Supports Drawing and Programming

The notion of flow is one possible type of semantic information that can be used to extend the implicit model of graphical rewrite rules. An immense range of problems can be represented as flow. The use of flow as semantics is discussed in the context of the Agentsheets [12] programming substrate. Agentsheets is used to create domain-oriented visual programming and simulation environments.

Despite their former introduction to the Agentsheets environment in 1991, rewrite rules, because of the shortcomings discussed in the previous section, did not play an important role. Visual programming languages such as the Voice Dialog Design Environment created for US WEST [11] were created in Agentsheets' textual programming language called AgenTalk.

An analysis of more than 40 applications implemented in Agentsheets revealed the use of Agentsheets as tools to create spatio-temporal metaphors including flow. Flow, with more than 50% of all Agentsheets applications using it, in one form or another, is the predominant metaphor used. Abstractly, flow in Agentsheets can be perceived as:

Flow = propagation of *agents* through a discrete space constrained by *conductors*

For instance, Figure 2 shows an agentsheet containing agents representing a pump,  , which creates water agents who are constrained with respect to where they can go by pipes,  , serving as conductors.

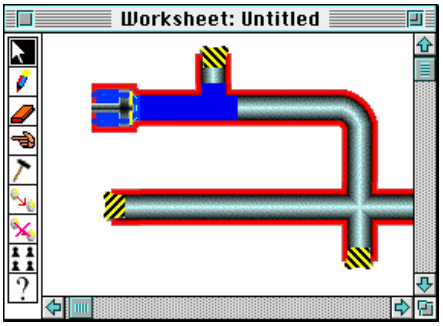

Figure 2. A flow system

Other kinds of conductors are: wires,  , rivers,  , roads,  , railway tracks,  , and conveyor belts  . Agents propagated can represent entities such as electrons, water, cars,  , and trains,  .

The following subsections describe how flow can be represented and the benefits that result from doing so in the context of a scenario creating a simple cars-and-roads world.

2.1. Defining Object Look: Icon Bending

One way to represent flow semantics is by augmenting objects with *connectivity* [9]. Connectivity of an object specifies how it connects its four neighbors. That is, whether the objects send output or receive input from neighbors.

The scenario starts by drawing a single horizontal road segment,  . The Agentsheets connectivity editor (Figure 3) is invoked to augment the road segment with arrows representing input and output connectivity.

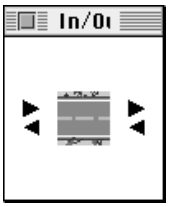

Figure 3. Road connecting neighbors left and right

The road segment is defined to connect things on the left of the segment with things on the right of the segment and vice versa. Arrows pointing towards the road indicate inputs, arrows pointing away from the road indicate outputs.

The Agentsheets gallery (Figure 4) is a repository of agent depictions. Applying a flow-specific icon transformation script to the road segment will create an entire *family* of road segment variations by transforming their look as well as their connectivity pattern (details of these transformations can be found in [9]). The links between road segments indicate family relationship and transformation.

These *syntactic and semantic icon transformations* are versatile in that they can be applied to a wide variety of objects representing conductors of any kind. The amount of time saved to draw individual objects is substantial. Hours of painstaking icon drawing can be reduced to seconds.

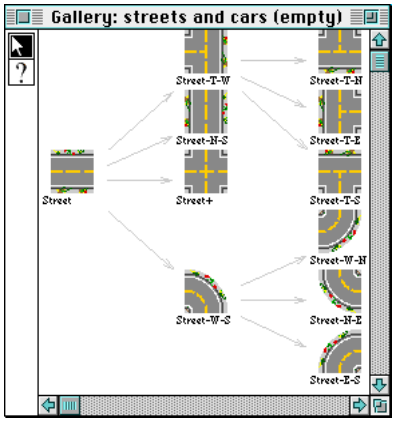

Figure 4. Road segment transformed in gallery

2.2. Laying Out Dependent Objects: Automatic Layout

The next step is to lay out dependent objects into a consistent scene. In order to draw a traffic situation road segments could be selected in the gallery (Figure 4) and put into an Agentsheets worksheet one by one. This approach is tedious because it requires users to select the right kind of segment to fit each specific situation.

Connectivity is used as input to the *automatic layout* mechanisms. Instead of selecting the specific road segment in the gallery, a user selects only the most general depiction (our original road) and draws with the draw tool,  , into the worksheet. According to their connectivity conductors will be matched up with surrounding conductors.

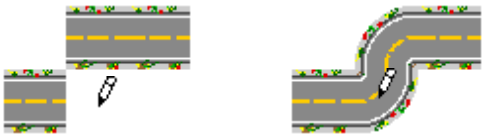

Figure 5. A new road segment gets added

A new road segment is about to be dropped in Figure 5 (left). To drop the road segment will not only drop the correct segment at the location of the draw tool but will also update all four potential neighbors because they may need to be changed in order to reflect the change. In Figure 5 (right) the road segment above the tool position has changed to a curved segment. The automatic layout mechanism[1] makes it possible to efficiently draw complex scenes.

Another important benefit resulting from automatic layout is consistency resulting in the simplification of rewrite rules. The creation of a situations in which input and output are not matched up (e.g., Figure 6) is avoided.

---

[1]This automatic layout mechanism can be perceived as a special case of a rewrite rule.

Figure 6. Road input/outputs are not matched up

As we shall see in the next section, automatic layout helps to prevent combinatorial explosion in rewrite rules because rewrite rules can rely on consistent layouts.

2.3. Defining Object Behavior: Semantically Enriched Rewrite Rules

Connectivity is one form of semantic enrichment of graphical rewrite rules that help to avoid combinatorial explosion. Using connectivity, new objects that get introduced after rewrite rules have been defined can be interpreted by the Agentsheets rewrite rule interpreter.

2.3.1. Avoiding the Combinatorial Explosion

How can rewrite rules be used to make a car follow roads? First a car gets introduced into the gallery (Figure 4) and then it is dropped into a worksheet (Figure 7):

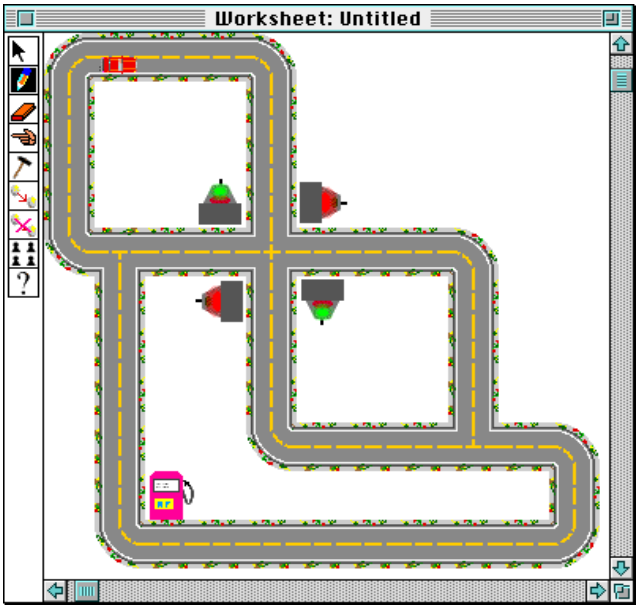

Figure 7. Car in road system

Double clicking the car brings up the Agentsheets rewrite rule editor (Figure 8).

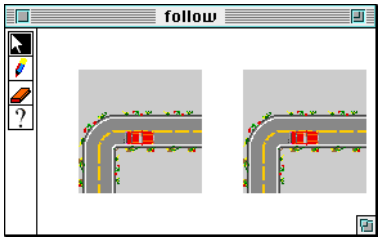

Figure 8. Agentsheets rewrite rule editor

Rules are attached to objects. Initially both sides of the rule describe the situation the object is in at the moment of editing. The right-hand side is edited to describe the future situation. The car could simply be moved one position to the right to represent the desired behavior. However, this literal rule would only apply in this very specific situation and would not be of a very general nature.

The general behavior of *following* can be expressed as the relationship between a car and two road segments.

> A car placed on a road segment X can move to a road segment Y if (i) X and Y are adjacent and (ii) one output of X lines up with an input of Y. (ii) implies (i)

Even this simple scenario, in which we do not deal with additional conditions such as the presence of traffic lights the literalness of conventional syntactic rewrite rules results in *combinatorial explosion*. There are 4096 possible situations in which cars could be in the context of 2 adjacent road segments. In this orthogonal world cars have 4 directions and adjacency is possible in 4 directions. If we think of road segments as objects connecting up to 4 neighbors we get 16 different looking segments. Finally we end up with 4 x 4 x 16 x 16= 4096 combinations resulting from the 4 car directions, 4 adjacency directions, 16 variations of road segments the car is on, and 16 variations of road segments the car moves to.

This is a very pessimistic way to determine the number of rules used to define the follow behavior. The number can be significantly reduced because no rules need to be defined for situations in which the car is not supposed to move, and for situations that are unlikely for the car to get into. Also the complete set of road segments includes 4 different dead ends (1 input and output), and 1 dead spot (0 inputs/outputs), which can be ignored for many flow-related applications. By slightly changing the problem, the number of rules required can be reduced further but still an end user is left with the need to define seemingly redundant rules that could have been derived by the system.

In Agentsheets the necessary context can be further reduced to a single road segment because of the automatic layout procedure. This procedure prevents the misalignment of adjacent conductors. Rewrite rules can safely assume that a

car cannot end up in a situation such as the one shown in Figure 9.

Figure 9. Automatic layout procedure eliminates this kind of situation

Therefore, rewrite rules do not require a second road segment. The rewrite rule shown in Figure 10 makes the car move to the right onto another valid road segment.

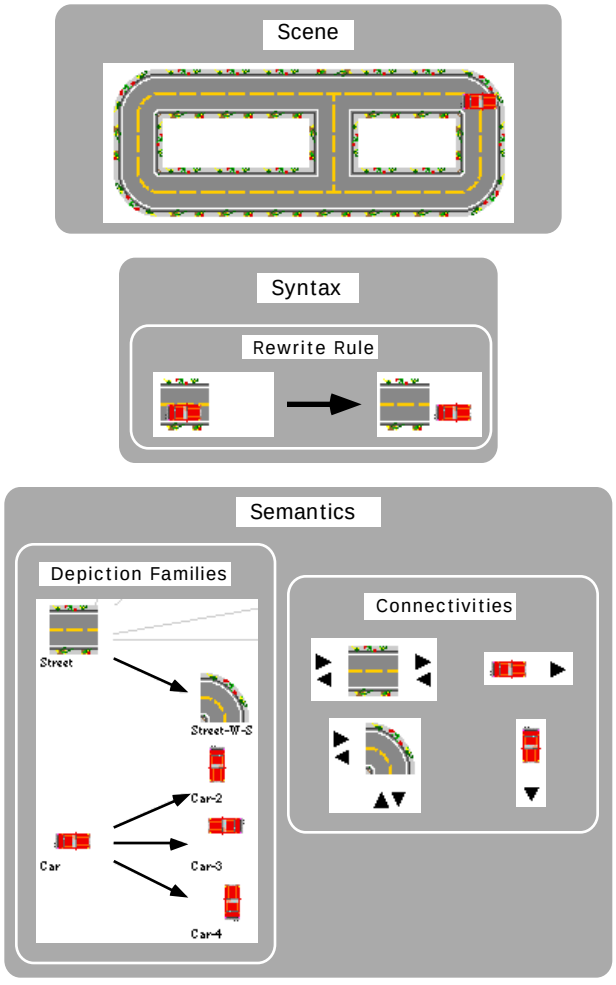

Figure 10. Interpretation based on syntax and semantics

The combination of the rewrite rule and semantic information specified by the user is sufficient to describe **all valid moves** from one road piece to another. If possible, the rule interpreter literally interprets rewrite rules. The literal interpretation of the follow rule (Figure 10) allows the car to move on the straight horizontal road all the way up to the curve.

Faced with a situation that does not literally match with any defined rewrite rule the rule interpretation mechanism needs to shift from the syntactic *literal interpretation* to a semantic *flow-oriented interpretation* that considers connectivity and depiction family membership:

Literal Interpretation	*Flow-oriented Interpretation (road = conductor, car = agent)*
if exactly this car is on exactly this road then it moves to the right	if *some* car is on *some* road then it moves toward an output of the road

The road segment under the car is tested for family membership. Is the  found in the scene related to the  found in the rule? Since this is the case and also since the car found in the scene and the car defined in the rule are from the same depiction family the car can move towards an output. A curve has two outputs. A *minimal rotation* heuristic is used to select the more plausible option. Turning back would be a 180-degree rotation. Moving down is only a 90 degree rotation. Moving down is chosen. In essence, one could say that the exhibited behavior is the result of *bending the rule* defined by the end user.

The car gets *aligned* with the road. The car is annotated with direction. Cars facing in 4 directions have been created with a transformation script. This sense of direction is given to the car to make it look “right” when moving in different directions. In the scenario the car also gets rotated 90 degrees when moved down.

If there are multiple minimal rotation options one is selected at random. For instance, when a car heading east ends up on a  road segment it will turn with a 50-50 chance north or south.

2.3.2 New Behavior From Old Rules

The road segments previously defined in the gallery (Figure 4) represent only 11 out of 16 possible conductors. Depending on the kind of flow is represented, it can make sense to define some of the remaining conductors. In our scenario the end user wishes to introduce dead end roads to have an explicit representation of the end of the road and to make cars turn around.

A new depiction representing a dead end is added to the road segment family in the gallery. It is edited to have the appearance of a dead end road (Figure 11).

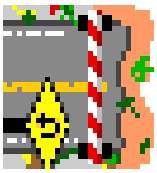

Figure 11. Dead end

The user defines the dead end to have only one input and output on the left (Figure 12).

Figure 12. Dead end connectivity

In the gallery a transformation script is used to create the remaining 3 dead ends by rotating their look and connectivities (Figure 13).

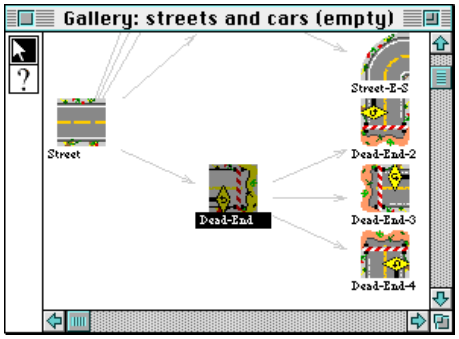

Figure 13. Transformed dead end depiction family

These additional definitions of semantics are immediately usable by the automatic layout mechanism. For instance, deleting the road segment under the eraser, , in Figure 14 (left) leads to the new situation in Figure 14 (right).

Figure 14. Automatic layout with dead ends

Without having to edit any rewrite rules the new road segments implement the desired behavior by defining the appropriate connectivity. Whenever a car encounters a dead end it will turn around and move back to where it came from.

2.4. Reusing Behavior: Rewrite Rule Analogies

Semantic information such as connectivity can be employed in a mechanism for *reuse by analogy*, where it acts to support a pupstitution [1, 6] process. If rules have been created to define how cars move on roads, then it should be possible to use an analogy to create a corresponding set of rewrite rules to define how trains move on tracks. This process is illustrated below.

First, trains and train tracks are added to the Agentsheets gallery. The track is then transformed via the same syntactic and semantic transformation script that was originally applied to the road. Again, the train is given a direction and all four rotations are generated (Figure 15).

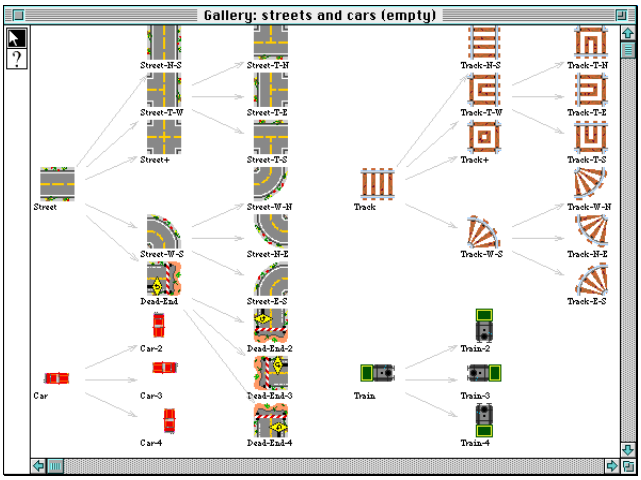

Figure 15. Isomorphic depiction families

At this point one could create the behavior for the train from scratch. Or, one could argue that a train basically follows a track like a car follows a road. A rewrite rule analogy is used to express this relationship (Figure 16). Since, at this point in the scenario only a single rule has been defined for the car, programming from scratch would be quite feasible or even preferable. However, in the more general case where there is a larger number of more complex rules, an approach based on analogy becomes advantageous.

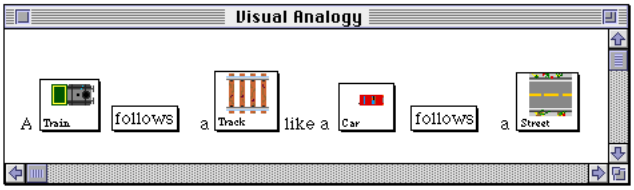

Figure 16. Analogy Editor

In contrast to object-oriented programming the rewrite rule analogy could be considered a *verb-oriented programming* approach. In rewrite rule analogies, the things to reuse and refine are not single objects but behavioral relationships between sets of objects. Here, the user specifies that a train's movement is constrained by a track in the same way that a car's movement is constrained by a road. Through the use of iconic pop-up dialogs, the user establishes this analogous mapping.

The result of the analogy is an instantiated set of graphical rewrite rules in which the cars and roads have been substituted with trains and tracks. The analogy would be inflexible and of very limited usefulness if the system understood only the literal mapping between , and . The semantics provided by connectivity help to overcome this type of literalism as well. Connectivity is used by the system to establish the mapping between any two cars/trains and roads/tracks. The system now infers that are like . Consequently, the new rewrite rules specifying how trains move through curved tracks are inferred by the analogous rules that tell the system how a car follows curved roads.

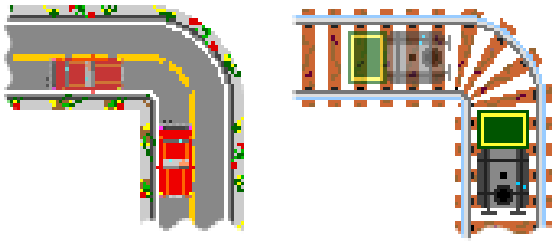

Figure 17. Trains follow tracks like cars follow roads

3. Discussion

The semantically enriched rewrite rule framework presented is not the solution of all the problems arising from the use of rewrite rules. The metaphor of flow and its representation through connectivity is just one form of semantics to extend the implicit model of graphical rewrite rules. The general applicability of semantics to rewrite rules or other end-user programming approaches is not only limited by technical feasibility but also by people's mind sets. Flow, for instance, can be applied to an extremely wide range of applications. A simple pacman game can be implemented using flow. Pacmen , and monsters, , are the flow agents moving through a maze consisting of hallway segments, , serving as conductors. Once the mapping between a problem like the pacman game and the flow metaphor is established, the solution becomes trivial. However, to recognize this mapping can be problematic.

The following subsections discuss very briefly other approaches to overcome the problem of combinatorial explosion without the addition of semantic information.

3.1. Topological Rule Interpretation

Topological rule interpretation is more resilient with respect to spatial transformations. In contrast to Euclidean representations, properties captured in topological representations, such as containment and connectivity, remain unchanged when scenes are changed via operations including rotation or reflection. In a graphical rewrite rule system based on topological spatial relations such as ChemTrains [2], the solution of a simplified version of the problem becomes trivial. A loop of road segment gets defined to be the places that can contain cars. These places are linked with arrows to represent connected road segments (Figure 18). Finally, a car gets defined and dropped into one of the road places and programmed via a single rewrite rule (Figure 19) specifying that a car simply moves from one place to another by following a link.

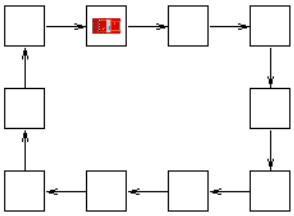

Figure 18. Topological road representation

Figure 19. Topological rewrite rule

The simplicity of the solution, however, may come at the price of abstraction. In contrast to Euclidean interpretation of rewrite rules, topological interpretation affords greater representational flexibility. Road segments can be put anywhere in scenes, and relationships between them can be expressed by simply connecting any pair of road segments. Unlike connectivity in Agentsheets, connectivity in ChemTrains becomes an explicit part of scenes and rewrite rules. On the one hand, putting this kind of information into scenes is ideal for explicit spatial notations such as graphs in which arrows are expected to be seen by users. In applications such as the car and road application, on the other hand, explicit spatial notations may lead to unwanted abstraction of the scene through the need to represent Euclidean relationships such as left and right with topological primitives such as connectivity.

3.2. Rotation and Symmetry

Siromoney et al.[13] extended Kirsch's rewrite rule model with reflection and rotation. Furnas included these

transformations as attributes to rewrite rules in his BitPict [4] system. The application of these transformation to entire situations consisting of complex objects is less obvious. If the situations to be transformed consist of orientation free objects, such as pixels, then the transformation of individual objects is irrelevant. However, if objects have their own discernible orientation, then scene transformation needs to operate at the level of scenes and objects. Unfortunately, it is not always desirable to transform all or any objects if a scene gets transformed. Transformation mechanisms will probably need user input to make these kind of decisions. Nonetheless, rotation and reflection are powerful approaches to reduce the number of rewrite rules required and should be used when semantic approaches are not applicable.

3.3. Sparse Scenes

A different approach to avoid combinatorial explosion is by making the construction space more sparse. Smith [14] proposes an elegant simplification of the car-on-road problem. The number of rewrite rules can be dramatically reduced by disallowing adjacency of complex road segments. Complex road segments are all non-straight road segments. If there always is a straight segment between any two complex segments, then the number of rules drops to 48. Every rewrite rule defines the movement of a car from a straight piece via a complex segment to another straight segment. A consequence of this approach is that cars dropped onto a complex segment would not know what to do because there are no corresponding rewrite rules. To guarantee sparse layouts a modified version of the automatic layout mechanism discussed may be of help.

4. Conclusion

Steps toward a semantically enriched model of graphical rewrite rules have been presented. The process of acquiring semantic information from end users to describe object characteristics needs to be understood from an economical perspective. End users are most likely to provide this kind of additional information when they perceive clear benefits. In the case of Agentsheets, these benefits include support for defining object look, laying out scenes consisting of mutually dependent objects, defining non-trivial behavior with a significantly reduced number of rewrite rules, and reusing existing behaviors via analogies.

Connectivity is an extension of the Euclidean rule interpretation model, that is common to most grid-based rewrite environments, with *topological* interpretation. The ability to control connectivity allows users to get the best of both worlds.

5. Acknowledgments

The research was supported by NSF (No. RED 925-3425 & Supplement), ARPA (No. CDA-940860), and Apple Computer Inc. Corrina Perone implemented the rewrite rule analogies, Carol Marra helped to build the Agentsheets rewrite rules. Many thanks for suggestions and discussions to Jim Ambach, Brigham Bell, Roland Hübscher, Clayton Lewis, Corrina Perrone, David Smith, Markus Stolze, and Tamara Sumner.

6. Bibliography

1. Anderson, J. R. and R. Thompson, "Use of Analogy in a Production System Architecture," Paper presented at the *Illinois Workshop on Similarity and Analogy*, Champaign-Urbana, 1986.

2. Bell, B. and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," *1993 IEEE Workshop on Visual Languages*, Bergen, Norway, 1993, pp. 188-195.

3. Bell, B., J. Rieman and C. Lewis, "Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough," *CHI'91*, New Orleans, LA, 1991, pp. 7-12.

4. Furnas, G. W., "New Graphical Reasoning Models for Understanding Graphical Interfaces," *Proceedings CHI'91*, New Orleans, LA, 1991, pp. 71-78.

5. Kirsch, R., A., "Computer Interpretation of English and Text and Picture Patterns," *IEEE Transactions on Electronic Computers*, Vol. 13, pp. 363-376, 1964.

6. Lewis, C., "Some Learnability Results for Analogical Generalization," *Technical Report*, CU-CS-384-88, University of Colorado, Computer Science Department, 1988.

7. McIntyre, D. W. and E. P. Glinert, "Visual Tools for Generating Iconic Programming Environments," *Proceedings of the 1992 IEEE Workshop on Visual Languages*, Seattle, 1992, pp. 162-168.

8. Nardi, B., *A Small Matter of Programming*, MIT Press, Cambridge, MA, 1993.

9. Repenning, A., "Bending Icons: Syntactic and Semantic Transformation of Icons," *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, 1994, pp. 296-303.

10. Repenning, A., "Designing Domain-Oriented Visual End User Programming Environments," *Journal of*

Interactive Learning Environments, Special Issue on End-User Environments, Vol. 4, pp. 45-74 1994.

11. Repenning, A. and T. Sumner, "Using Agentsheets to Create a Voice Dialog Design Environment," *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, Kansas City, MO, 1992, pp. 1199-1207.

12. Repenning, A. and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *Computer*, Vol. 28, pp. 17-25, 1995.

13. Siromoney, G., R. Siromoney and K. Krithivasan, "Picture Languages with Array Rewriting Rules," *Information and Control*, pp. 447-470, 1973.

14. Smith, D. C., Cars 'n Roads in KidSim, personal communication, 1995

15. Smith, D. C., A. Cypher and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, Vol. 37, pp. 54-68, 1994.