Composing Complex Behavior from Simple Visual Descriptions*

Roland Hübscher
EduTech Institute & College of Computing
Georgia Institute of Technology
Atlanta, GA 30319–0280
roland@cc.gatech.edu

Abstract

An often-mentioned advantage of rule-based programming languages is that a program can be extended simply by adding a few more rules. In practice however, the rules tend to be dependent on each other and instead of just adding rules, existing rules need to be changed. The unique rules in *Cartoonist*, a rule-based visual programming environment to build simulations, provides a solution to this problem. Cartoonist's rules can be used in a more modular way supporting an iterative mode of programming. Libraries of visual descriptions can be built and reused to compose complex behavior from these descriptions. This makes exploring the space of possible descriptions of simulations easier, which is valuable for intended educational use of Cartoonist. Another advantage of Cartoonist is that its programs tend to have fewer and simpler rules than programs written for comparable systems.

1 Introduction

Cartoonist, a rule-based visual programming environment to build simulations [7], was designed to be used by students up to the 12th grade to create models and run them as simulations. The students must be able to easily explore the space of alternative models without getting bogged down by the complexity of the programming language.

Exploring alternative models requires the description of a model to be modular so that the behavior of the objects in the simulations can easily be extended or changed. A modular approach also supports an iterative programming style, where the student can refine the behavior of the objects by adding more and more rules to the description of the behavior.

All the existing rule-based visual programming systems including Cocoa (formerly called KidSim) [9], Science Theater, Agentsheets [8], ChemTrains [1], and BitPict [5] are based on production systems using rewrite rules [3, 6]. Although, it is sometimes claimed that these rule-based systems can be extended by simply adding rules, this does not always work in practice. Rules can interact with each other in subtle ways and adding new rules often requires some existing ones to change in non-trivial ways.

This limited independence of rewrite-rules, also found in visual rule-based systems, can cause problems when the user wants to change a simulation slightly or reuse parts of another simulation.

Cartoonist's *constraint rules* provide an approach to rule-based visual programming that, among other things, solves the problem of modularity of a large and interesting class of simulations. These simulations can be described in terms of combinations of simple behavioral patterns of the objects in the simulation. In general, each of these behavioral patterns can be implemented with one constraint rule, and adding or changing behavioral patterns can be done without having to change other rules.

*Published in the Proceedings of the 1993 IEEE Symposium on Visual Languages, pp. 88–94.

Cartoonist shares several characteristics with most of the mentioned visual rule-based systems. All use grid-based representations with the exception of ChemTrains, and use objects that can move around and change their iconic appearance. Cartoonist calls these objects *characters*.

Cartoonist differs from the other systems in several ways that will be discussed in more detail, later. As mentioned earlier, the other systems use rewrite rules consisting of a before- and an after-picture. The rewrite rule is applied to the simulation display (which is what the user sees while running the simulation) by replacing the before-picture with the after-picture. The main differences between the rule-based systems and Cartoonist are shown in the following list.

- Cartoonist provides the user with a collection of characters already associated with a set of actions. For instance, a character can move in any of the four directions if the destination is empty. In the other systems, the characters have no built-in actions.

- A constraint rule describes what states can follow each other. Whereas a rewrite rule generates a state, a constraint rule selects a that is generated by the characters' actions. Thus, programming with constraint rules means organizing the characters' actions leading to the appropriate behavior of the characters.

- Constraint rules can access any number of past simulation states in their condition.

- In Cartoonist, negation can be used in any constraint of a rule, even in the one describing the after-picture.

- Several constraint rules can be used to find the "best" action if more than one constraint rule applies. Using the voting schema described in detail below, the behavioral patterns implemented by the rules are combined dynamically.

Figure 1: A rewrite rule making a ball (or person seen from above) going to the right. The gray square means that it must be empty.

In this paper, Cartoonist's constraint rules will be described and compared to the rewrite rules used in visual programming systems. Then, a realistic problem (simulating people walking around in an airport) will show some of Cartoonist's strengths. Rewrite rule-based solutions are suggested as comparisons. Then, the reasoning engine is explained showing how Cartoonist actually accomplishes the results shown. In conclusion, the results will be summarized.

2 Visual Rule-Based Programs

In all of the visual rule-based systems, including Cartoonist, a program consists of characters and rules defining the behavior of the characters. Whereas all the other systems use *rewrite* rules, Cartoonist uses a similar looking, yet quite different type of rule that is sometimes also called a cartoon or a constraint rule.

2.1 Rewrite Rules

A rewrite rule consists of a before- and an after-picture. If the before-picture is consistent with a part of the simulation display, then the rule can be applied by replacing the matched part in the display with the after picture. For instance, the rewrite rule in Figure 1 implements a ball moving to the right. Most systems allow one to automatically generalize such a rule in all four directions in the grid.

As a second example, we describe a person that keeps going in the same direction. The rewrite rule in Figure 1 would not work since it does not describe where the person came from. A simple way to solve this problem is adding some kind of directional feature, for instance a long nose that points into the direction the person is walking.

In many situations, more than one rule could be applied since several before-pictures are consistent with

Figure 2: A constraint rule with three constraints.

Figure 3: Using negation in the after-picture of a constraint rule.

different parts of the simulation display. In most visual rule-based systems, the rules are in some user-specified order and the first rule in this list is applied whose before-picture is consistent with the simulation display. Generally, the more specific rules are kept at the beginning of the list and the more general catch-all rules are moved towards the end.

2.2 Cartoonist's Constraint Rules

As mentioned earlier, the characters in Cartoonist are already associated with actions. For instance, the people can move in any of the four directions if the destination square is empty. A palette of characters with the actions are initially provided to the user. If required, the user can extend the palette with new characters.

The constraint rule in Figure 2 shows that a rule in Cartoonist can also use a past state of the simulation in its before-picture which is useful for describing simulations. The last of the three pictures corresponds to the after-picture in a rewrite rule, the second-to-last picture always refers to the present state of the simulation and the pictures on its left refer to past states. Thus, the rule in Figure 2 describes a person coming from the left and going to the right. In Cartoonist, rules can be automatically generalized into all four directions such that the person keeps walking straight no matter what the initial direction was.

Cartoonist's constraint rules can use negation also in the after-picture because all the pictures referring to the past, present and future are constraints. For instance, the rule in Figure 3 says that the person displayed in the before-picture will *not* be at the same place in the next state. A crossed-out character means that this character is not at this place. Describing what should *not* happen is another powerful tool provided by Cartoonist to describe simulations.

Similar to the other rule-based visual systems, the rules in Cartoonist are sorted using a priority value, where several rules may have the same priority value.

3 Using Constraint Rules

An example will demonstrate how constraint rules are used to describe and combine behavioral patterns. The implementation of the scenario with rewrite rules is also discussed

3.1 A Scenario

Assume that, in the context of a larger problem, the 6th grade students need to study the behavior of a group of people. These groups of people show the following behavior:

- they never rest;
- they try hard to avoid each other such that they never touch each other;
- once they walk in a direction they stick to it (under the condition that they avoid each other);
- when they run into a wall they seem to bounce back like balls (again under the condition that they can avoid each other).

This, admittedly strange, behavior can often be observed in airports—maybe with the exception of the last part of the description.

In the simulation, the people are put into a small room shown in Figure 4. They can walk in any of the four directions in a grid which is the visual representation used in most of the rule-based visual programming systems.

This description of these people sounds simple, yet describing a behavior (walking straight, turning around at walls) under the constraint of avoiding each other is difficult for rewrite rule based systems because the behaviors interact in a non-trivial way.

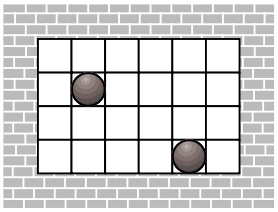


Figure 4: Two persons walking around in a small room.

Figure 5: Never stand still: the person in the before-picture will not be at the same place in the after-picture.

Once the rewrite rules are in place, the description of the people's behavior is difficult to change.

These are the kinds of problems in which Cartoonist's unique constraint rules show their strength. Since each of the four parts of the behavior can be independently described, adding yet another constraint is trivial.

3.2 Cartoonist's Solution

The behavior of the people is described with four behavioral patterns. For each of these patterns, one constraint rule is required. They can be developed separately and even stored in a library which simplifies building similar simulations.

The first part of the description states that the people never stand still. This is expressed by the rule in Figure 5.

The second requirement that people avoid each other at all cost can again be expressed using negation in the after-picture. Figure 6 says that the person moving to the right (the person is the one at the left in the before-picture and the one in the middle in the after-picture) will not be next to any other person. This rule needs to be generalized in all four directions which is done automatically by Cartoonist.

The constraint rules in Figure 7 and Figure 8 take advantage of the fact that the rules may access past states. They describe the last two requirements in a similar fashion.

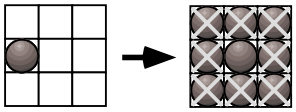


Figure 6: People avoid each other: whenever a person moves it must not end up next to any other person.

Figure 7: Keep going in the same direction.

Figure 8: Turn around at the wall.

Next, the priority values have to be assigned to the four rules. There are many ways of assigning the priority values as long as the avoid-each-other has a higher priority than the last two requirements. This follows from the constraint that the people have to avoid each other no matter what and only under this constraint they are allowed to walk straight or bounce off a wall they just ran into. The rest of the paper assumes that the first two rules are assigned the same priority and that a lower priority value is assigned to the latter two requirements.

3.3 Using Rewrite Rules

The rewrite rules take advantage of a directional feature, e.g., a nose, since they do not have the ability to access past states. There is no reason why rewrite rules could not be extended to have access to past states.

All of the four behavioral patterns can then be expressed by a set of rewrite rules. Not standing still is simple and the rule in Figure 1 generalized to all four directions implements this behavior. Walking straight and turning around at a wall can be similarly implemented with the nose as representation of the direction direction.

Avoiding each other can also be implemented with rewrite rules, however, a large number of rules is necessary. The resulting description is not nearly as elegant as the one with constraint rules which allow negation in the after-picture.

However, once these behaviors are programmed separately, there is no way that they can be combined just by arranging them in some order. The conditions of the rules must be explicitly changed so that the correct action is executed in a given situation. For instance, each move-straight rule and each turn-back-at-wall rule must check that the person does not end up next to another person. This leads to very many very complicated rules.

Rewrite rules can be used to implement the simulation, however, the solutions tend to be much more complicated than Cartoonist's and will be difficult to extend.

4 Cartoonist Engine

The reasoning mechanism deciding which character executes what action is based on a voting mechanism. The constraint rules vote for some of the actions and the action with the "best" votes is executed. This section describes the details of the matching process, the interaction between the characters' actions and constraint rules, and the voting mechanism.

A *character* is an object in the simulation consisting of an icon and a set of actions. The icon determines the character's appearance. The set of actions determines what can be executed by the character, for instance, to move around or to play a tune. In all the examples in this paper, a wall has no action and a person can move east, west, north, or south. Persons can move only into empty squares, so they cannot move into a square occupied by another person or a wall.

In Figure 9, two persons are surrounded by a wall and therefore can not leave the 24 squares. Note that this is not dependent of the constraint rules the user will rewrite, but only on the actions a person can execute. The person on the left moved one square to the right and the person at the bottom moved one square up. The rest of the section discusses where the person at the top left is moving and why. I will refer to the state on the right in Figure 9 as the current state with its past shown in the left part of the same figure.

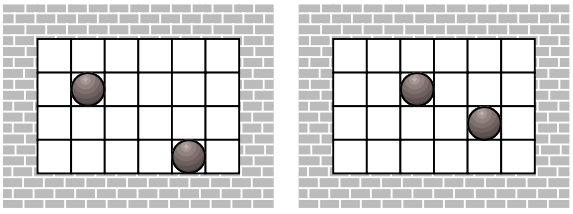


Figure 9: The situation on the left precedes the situation on the right, where the upper person moved right and the lower person moved up. The grid is useful for the discussion, but is not drawn in the actual simulation.

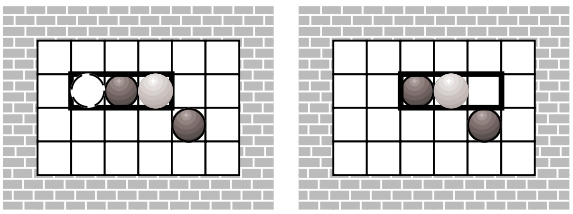


Figure 10: The left picture shows a voter with the previous person position (left) and the potential future position (right). The right picture shows a constraint instance that is not a voter. The potential future person position is also shown.

4.1 Voters

A constraint rule is of the form

$$c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_k \rightarrow c_{k+1}$$

where the c_i's are the constraints, that is, the pictures. Constraint c_k describes the current state, c_{k+1} a state in the immediate future, and $c_1, \ldots, c_{k-1}$ states in the past.

A rule instance is called a *voter for state s* if the constraints match a state sequence such that $c_1, c_2, \ldots, c_{k+1}$ describe a temporally contiguous sequence of states and c_{k+1} matches state s in the immediate future. It is this latter condition, that makes a rule instance a voter. Finding all the voters requires therefore to generate all the possible states in the future which can lead to an expensive look-ahead search.

A few examples will clarify the definition of a voter.

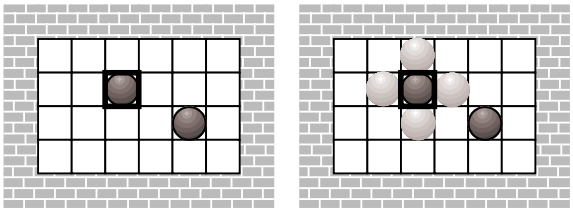


Figure 11: The left picture shows where the rule is matching and the right picture shows all the potential future person positions.

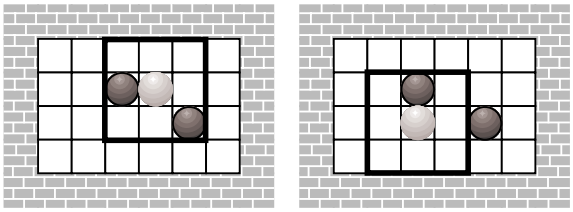


Figure 12: Two of the four ranges that have to be checked for the top left person and the avoid-each-other rule are marked. Moving to the right does not get a vote from this rule, as the left picture shows. The avoid rule voted in favor of the person moving down (and left and up).

Figure 10 shows the current state of the simulation and how the rule → → is matched against the state. In the left picture, the rule matches the former (empty disk/person), current (dark gray disk), and potentially future position (light gray disk) of the person. This voter directs the person to the right. The rule instance in the right picture shows is not a voter and thus, it does not vote for the person going to the right.

Figure 11 shows on the left how the rule → is matched against the person. It is also matched similarly against the other person. In the picture on the right, the potential positions of the person in the same figure are shown. Matching the rule as in the left picture leads to four voters, each voting for another action leading to the four potential locations (shown as light gray persons) in the right picture.

Figure 12 shows how the rule → is matched against the top left person. The picture on the left shows that the rule cannot vote in favor of the person going to the right because of the person at the bottom right. However, the picture on the right

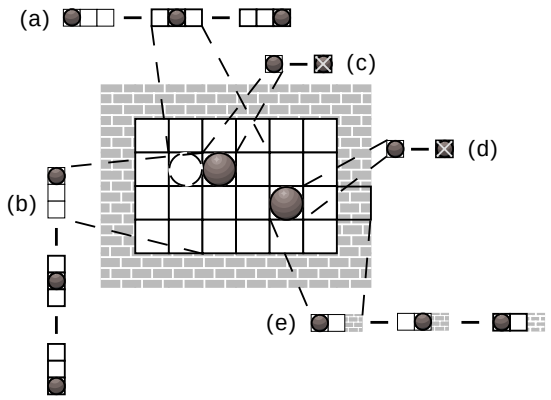


Figure 13: Some of the instantiated constraint rules are shown. For instance, the rule instance (a) is a voter whereas (b) is not.

shows a match that leads to a vote for the person going down one square. Going up and to the left will also get a vote from this rule. Because all the constraints of a rule must be used to be eligible to vote, and the last constraint describes the state in the immediate future, a look-ahead search of depth one is necessary to find the voters.

The current prototype is implemented on top of Agentsheets and employs an expensive look-ahead search using a temporally extended Rete network [4]. Current research suggest that the look-ahead search can be completely avoided using some more advanced techniques at compile time and by slightly constraining what actions the characters may execute.

Figure 13 shows a few voters and rule instances which are not voters because their last constraint does not match a state in the immediate future. Instance (a) is a voter because its second-to-last constraint describes the current state. It votes for the person moving to the right. Instances (b) and (e) are no voters. Finally, instances (c) and (d) vote for four actions each, namely, moving either person in any of the four directions.

4.2 Voting Scheme

The voting scheme implements the subsumption-like architecture [2] of Cartoonist. In the current state of the simulation, each action of each character gener-

ates a new state. Voting scheme is the algorithm that chooses one of these states as the next state.

The algorithm works the way the following example suggests. If you are hungry and are attacked by a tiger, the two behaviors, "run away" and "get some food," apply. Apparently, the former behavior is more important and suggests running in one of the directions east, west, or north. To figure out which one of the directions to take, the "get some food" behavior can be used, which suggests going to the south or the north. Because going north is consistent with the actions suggested by the "run away" behavior, you should run to the north. If the "get some food" behavior had suggested going the south only, then it could not have been used and one of the three actions suggested by "run away" would have been chosen in the absence of other applicable behaviors.

Each action generates a state for which constraints rule instances can vote. Each such state has a possibly empty set of voters that can be used to find the best state. The priority relation $\succ$ is defined such that $a_i \succ a_j$ means that voter a_i is preferred over voter a_j. A voter gets the same priority assigned as the rule constraint rule from which it is an instance.

The preference relation between voters can be extended to list of voters. A list of voters r_i is preferred over a list of voters r_j, if r_i's voters are preferred over r_j's voters, that is, if $r_i \succ r_j$, which is defined by $r_i \succ r_j \equiv v_1^i, \ldots, v_h^i \succ v_1^j, \ldots, v_k^j \equiv$

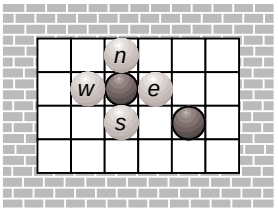
$$\exists l,\ l \leq h.\ \forall m,\ m < l.\ v_m^i \sim v_m^j \ \wedge\ \left(k < l\ \vee\ v_l^i \succ v_l^j\right)$$

The latter expression can be analyzed as to expressions each dealing with one of the two possible cases. The first expression shown below is used when the voters in r_j are just as good as in r_i but there are fewer voters in r_j than in r_i.

$$\exists l,\ l \leq h.\ \forall m,\ m < l.\ v_m^i \sim v_m^j\ \wedge\ k < l$$

The next expression takes care of the case where the first $l-1$ voters are pairwise indifferent and the l^{th} voter makes the difference:

$$\exists l,\ l \leq h.\ \forall m,\ m < l.\ v_m^i \sim v_m^j\ \wedge\ v_l^i \succ v_l^j$$



	e	w	n	s
(a)	v	v	v	v
(b)	x	v	v	v
(c)	v	i	i	i
(e)	x	x	i	x

Figure 14: The table at the bottom shows what votes the rules generate for the actions moving **e**ast, **w**est, **n**orth, and **s**outh. The letters in the table stand for "**v**oter," "rule **i**nstance," and an **x** stands for no instance at all. See the text for a more detailed discussion.

Finally, a state is chosen whose list of voter is the best, that is, there is no other state with a better list of voters. If there is or more than one best state, one of the best states is chosen randomly.

Figure 14 illustrates the voting scheme with the now familiar constraint rules. The priority values were assigned earlier to the rules and are repeated here. Let α_i and α_j be two rules. Then $\alpha_i \succ \alpha_j$ means that α_i has the higher priority than α_j. For all pairs of rules where the $\succ$ relation is not stated it can be assumed that the rules have the same priority. The priorities for the four rules of the scenario are summarized in Figure 15.

The action that the person at the top in Figure 9 is going to execute is found as follows. First, all the voters for each of the four actions e, w, n, and s are collected (see Figure 14). The columns of the table are named with the actions and the rows with the rules. Action e gets two votes from the voters generated by the rules on lines (a) and (c) and the other three actions get two votes each from the voters generated by the rules in rows (a) and (b). These sets of

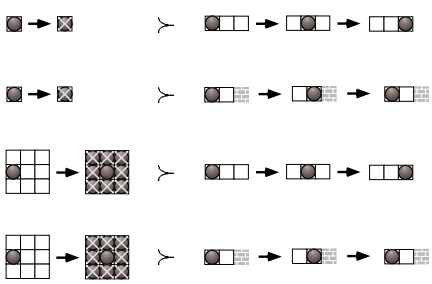


Figure 15: The don't-rest and the avoid-each-other rules have a higher priority than both of the other rules.

voters have to be compared according to the priority values of the voters.

A set V_i of votes is better than another set V_j of votes if V_i has more voters with higher priority values. The exact definition can be found in [7]. The action with the best set of votes will be executed, where the sets of votes for the actions are

e: { a, c }
w: { a, b }
n: { a, b }
s: { a, b }

The best set is found assuming the following rules.

1. Voters of equal importance neutralize each other.

2. If a set has a voter that is preferred over the best voter of another set, then the second set loses.

Applying the first rule leads to the sets

e: { c }
w: { b }
n: { b }
s: { b }

Then, the second rule gets rid of action e and the first rule leads to the final state

w: { }
n: { }
s: { }

None of these sets is preferred over another one, so one of the actions w, n, or will s be chosen randomly and executed.

5 Summary and Conclusions

Cartoonist's unique type of constraint rules can describe simulations by composing complex behavior from simpler visual descriptions. These rules can access the past states of the simulation and use negation in the after-picture which can considerably simplify describing of some types of behavior.

Cartoonist's unique rules advantages can be viewed from a programming language and from an educational perspective. First, a modular approach to programming is generally believed to be advantageous and that iterative programming is useful, at least in the prototyping phase which is the mode in which Cartoonist is always used. One problem of having global (constraint) rules is that some unwanted interaction between the rules might occur. This is not a problem in the examples shown because only one kind of behaving character, a person, was used in in each simulation. Therefore, it might be necessary for more complicated simulations to add a mechanism to group constraint rules.

Second, Cartoonist is intended to be used by students building models and exploring the space of alternative models. Reusing rules from other simulations is easier with Cartoonist than with rewrite rule-based programming systems. This also suggests that constraint rules may support collaborative design and the building of simulations, important activities for students.

References

[1] B. Bell and C. Lewis. Chemtrains: A language for creating behaving pictures. In *Proc. 1993 IEEE Symposium Visual Languages*, pages 188–195, Bergen, Norway, 1993.

[2] R. A. Brooks. Intelligence without reason. Technical Report 1293, Massachusetts Institute of Technology, A.I. Laboratory, 1991.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming in Expert Systems: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA, 1985.

[4] C. Forgy. Rete: A fast algorithm for many pattern / many object match problem. *Artificial Intelligence*, 19:17–37, 1982.

[5] G. Furnas. Formal models for imaginal deduction. In *Proceedings of the Twelve Annual Conference of the Cognitive Science Society*, pages 662–669, Hillsdale, NJ, July 1990. Lawrence Erlbaum.

[6] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat. *Building Expert Systems*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

[7] R. Hübscher. Visual programming with temporal constraints in a subsumption-like architecture. Technical Report CU-CS-778-95, Department of Computer Science, University of Colorado at Boulder, August 1995.

[8] A. Repenning and T. Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *Computer*, 28:17–25, 1995.

[9] D. C. Smith, A. Cypher, and J. Spohrer. KidSim: Programming agents without a programming language. *Communications of the ACM*, 37(7), 1994.