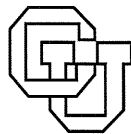


**Visual Programming with Temporal Constraints in a  
Subsumption-Like Architecture**

**Roland Hübcher**

**CU-CS-778-95**



**University of Colorado at Boulder**

**DEPARTMENT OF COMPUTER SCIENCE**



ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.



# Abstract

Visual rewrite rule-based programming languages have been shown to be an interesting new approach to end-user programming. These languages are used to create visual simulations in which a rewrite rule changes the simulation display by replacing a part of the display with a different picture. Although these languages show great promise, they have problems in describing easily a wide range of parallel behavior and refining iteratively the behavior of the objects in the simulation.

This dissertation introduces *Cartoonist*, an extension of rule-based systems based on the notion of *cartoons*, a generalization of rules. Each of the *characters*, the objects in the simulation, consist of a depiction and a set of actions that it can execute. A cartoon describes the behavior of these characters by specifying a state sequence that may extend arbitrarily far into the past. The behaviors are dynamically combined in a *subsumption-like* architecture.

*Cartoonist* embodies four main characteristics that distinguish it from rule-based systems in important ways.

- The complex behavior of the objects in the simulation can be composed from separate, simple descriptions.
- Cartoons can refer to the present and the past in their condition, simplifying the description of interactions of objects. They also provide a way to represent temporal concepts.
- Different kinds of parallel behaviors can be expressed.
- Due to the declarative nature of the forward-chaining cartoons, behavior can be described by specifying what should *not* happen as well as what should.



# Dedication

To my parents  
Berthe and Otto Hübscher-Zellweger





# Acknowledgments

I gratefully thank:

my advisor, Clayton Lewis, for being a great guide through the quicksand of research;

my thesis committee, Clayton Lewis, Wayne Citrin, Gerhard Fischer, Mark Gross, Jim Martin and Peter Polson, for their useful comments on earlier drafts of this document;

my best friend and wife Teresa Younger who wouldn't let me give up when I was tired and frustrated;

Alex Repenning for many invaluable discussions which helped shape this research;

Cathleen Wharton for giving me detailed and insightful comments on an earlier draft of this document;

my wonderful friends, Nadia and Alex Repenning, for their continued support;

Teresa Younger for helping make this document readable and understandable;

Brigham Bell, Cathleen Wharton, Markus Stolze, Kurt Schneider, Jim Ambach and Corrina Perrone for discussing of this work; and

the members of the NAP group and the participants of the Child's Play workshop for comments on this research and in particular Allen Cyphert for telling me what my research is all about.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Rule-Based Visual Programming Languages . . . . .	1
1.2	Context and Related Work . . . . .	3
1.2.1	Rule-Based Languages . . . . .	3
1.2.2	Declarative Languages . . . . .	4
1.2.2.1	Logic Programming . . . . .	5
1.2.2.2	Constraint Programming . . . . .	6
1.2.2.3	Planning . . . . .	6
1.2.3	Object-Oriented Programming . . . . .	7
1.2.4	Visual Programming Languages . . . . .	7
1.2.5	Programming by Example . . . . .	8
1.2.6	Behavior-Based Programming . . . . .	9
1.3	Organization of the Thesis . . . . .	9
<b>2</b>	<b>Cartoons</b>	<b>11</b>
2.1	Comic-Strip Metaphor . . . . .	11
2.2	Characters . . . . .	13
2.2.1	Rules: Conditional Actions . . . . .	14
2.2.2	Cartoons: Structuring Actions . . . . .	14
2.3	Visible and Invisible States . . . . .	15
2.4	Cartoons . . . . .	15
2.4.1	Cartoons Are Not Rules . . . . .	16
2.5	Voting Scheme . . . . .	16
2.5.1	Preferences between Cartoons . . . . .	17
2.5.2	Comparing Sets of Voters . . . . .	18
2.5.3	Subsumption-Like Architecture . . . . .	19
2.6	Illustration of the Voting Scheme . . . . .	19

2.6.1	Voters . . . . .	20
2.6.2	Voting Scheme . . . . .	22
2.7	Characteristics of Cartoons . . . . .	24
2.7.1	Referring to Past and Present in the Condition . . . . .	24
2.7.2	Using Emergent Relations to Describe the Future . . . . .	24
2.7.3	Declarative Forward-Chaining . . . . .	25
2.7.4	Partial Specification of Actions . . . . .	26
2.7.5	Flexible Description of Parallel Actions . . . . .	26
<b>3</b>	<b>Preview of the Cartoonist System</b>	<b>27</b>
3.1	Rule-Based Visual Programming Systems . . . . .	27
3.1.1	Rules . . . . .	27
3.1.2	BitPict . . . . .	28
3.1.3	ChemTrains . . . . .	28
3.1.4	KidSim . . . . .	29
3.1.5	Agentsheets . . . . .	31
3.1.6	ScienceShows . . . . .	32
3.2	Cartoonist Instances . . . . .	32
3.3	Agentsheets-Cartoonist . . . . .	34
3.4	Examples of Simulations . . . . .	37
3.4.1	Rolling Balls . . . . .	37
3.4.1.1	Attempts with Rules . . . . .	37
3.4.1.2	Solution with Cartoons . . . . .	40
3.4.1.3	Summary . . . . .	41
3.4.2	Game of Life . . . . .	41
3.4.3	Melting Ice . . . . .	43
3.4.3.1	Summary . . . . .	44
3.4.4	Collisions . . . . .	45
3.4.4.1	Summary . . . . .	46
3.4.5	Negative Programming: Avoiding Each Other . . . . .	46
3.4.5.1	Summary . . . . .	49
3.4.6	Pachinko Task . . . . .	49
3.4.6.1	Summary . . . . .	51
3.4.7	PacMan . . . . .	51
3.4.7.1	Solution with Rules . . . . .	52
3.4.7.2	Solution with Cartoons . . . . .	54

3.4.7.3	Summary . . . . .	55
3.4.8	Examples: Conclusions . . . . .	55
<b>4</b>	<b>Parallel Actions</b>	<b>57</b>
4.1	Parallelism in Rule-Based Systems . . . . .	57
4.1.1	Executing Actions in Parallel . . . . .	57
4.1.2	Increasing Efficiency . . . . .	60
4.1.3	Increasing Expressiveness . . . . .	61
4.2	General Framework for Rule-Based Systems . . . . .	62
4.2.1	Sequential Systems . . . . .	64
4.2.2	Parallel Systems . . . . .	65
4.3	Cartoonist Approach . . . . .	66
4.3.1	Other Parallel Behavior . . . . .	67
4.3.2	Comparing the Different Approaches . . . . .	69
4.3.2.1	Sequential: Random Action . . . . .	71
4.3.2.2	Sequential: First Action . . . . .	72
4.3.2.3	Parallel: All Actions . . . . .	73
4.3.2.4	Parallel: Test Before Execute . . . . .	74
4.3.2.5	Parallel: $\text{Enable}(a, s) \neq \emptyset$ . . . . .	74
4.3.3	Parallel Actions in Cartoonist . . . . .	74
4.4	Time Complexity . . . . .	75
<b>5</b>	<b>Implementation of the Cartoonist Prototype</b>	<b>79</b>
5.1	Cartoonist Algorithm . . . . .	79
5.2	Transforming Cartoons into Rules . . . . .	81
5.3	Interface between Cartoonist and Agentsheets . . . . .	82
<b>6</b>	<b>Conclusions</b>	<b>85</b>
6.1	Related Work Revisited . . . . .	85
6.1.1	Rule-Based Languages . . . . .	85
6.1.2	Declarative Languages . . . . .	85
6.1.2.1	Logic Programming . . . . .	86
6.1.2.2	Constraint Programming . . . . .	86
6.1.2.3	Planning . . . . .	86
6.1.3	Object-Oriented Programming . . . . .	87
6.1.4	Visual Programming Languages . . . . .	87

6.1.5	Programming by Example . . . . .	88
6.1.6	Programming with Temporal Information . . . . .	88
6.1.7	Behavior-Based Programming . . . . .	88
6.2	Results . . . . .	88
6.2.1	Retaining the Advantages of Rule-Based Systems . . . . .	89
6.2.2	Composing Behaviors . . . . .	89
6.2.3	Temporal Concepts . . . . .	89
6.2.4	Parallel Behavior . . . . .	89
6.2.5	Negative Programming . . . . .	90
6.3	Future Work . . . . .	90
<b>A</b>	<b>A More General Treatment of Temporal Constraints</b>	<b>99</b>
A.1	Temporal Constraints . . . . .	99
A.1.1	Formal Definition . . . . .	100
A.1.2	Temporal Operators and Invisible States . . . . .	101
A.1.3	Representation of Temporal Concepts . . . . .	102
A.2	Cartoons as Temporal Constraints . . . . .	104
A.3	Subsumption Architectures . . . . .	106
A.4	Preference Scheme . . . . .	106
A.4.1	Necessary Characteristics . . . . .	107
A.4.2	Formal Definition . . . . .	107
A.4.3	Discussion of the Preference Relations . . . . .	109
A.4.4	Preference Relation for Cartoons . . . . .	110
A.5	Subsumption Hierarchy of Cartoons . . . . .	111
A.5.1	Subsumption Conditions . . . . .	112
A.5.2	Avoiding Combinatorial Explosion with Cartoon Subsumption . . . . .	114
A.6	Summary . . . . .	115

# Chapter 1

## Introduction

Visual simulations can be useful tools in a grade-school science class, but not if programming them is difficult. End users, people who are not inherently interested in programming and computers, do not necessarily want to learn to program, because for them that task tends to be time consuming, difficult, and boring. New ways must be found to provide users with tools that simplify the creation of non-trivial simulations and to free the users from the necessity of learning to program.

Many new approaches to programming are concerned with eliminating the need for end users to spend time learning to program. The tools provided to the end user should help her to solve *actual* problems, that is, to reduce the complexity of the real problem without adding a disproportionate amount of complexity due to the use of the tool.

The *Cartoonist framework* for visual programming languages to create simulation is introduced in this dissertation. It is based on rule-based visual languages and increases the reasoning power of the underlying system without reflecting this internal complexity in the use of the system. This leads to a system for end users to visually program graphical simulations that could be found, for instance, in grade-school science classes. A Cartoonist system is used in a similar fashion as a purely rule-based system, yet it provides the user with a more powerful means to create simulations. Parallel actions can be described easily and the behavior of the characters in the simulation can be refined incrementally.

### 1.1 Rule-Based Visual Programming Languages

Using Cartoonist, an end user, defined here as a person who is not interested in computers and their programming but in their utility [57], can create and run visual simulations.

This definition of end user includes all kinds of people, from grade-school students to scientists to the casual home computer user. People who already know how to program in a

language such as Pascal are not necessarily excluded because they may dislike using Pascal but were forced into learning it due to unfortunate circumstances. Although the definition of end user covers a wide range of users, the examples that are presented are taken from a science class in grade school.

Visual rule-based programming systems have proven to be easy for end users to use [72, 5]. These systems can be used to create visual simulations by transforming the picture with the rules. A visual rewrite-rule consists of a before-picture and an after-picture. If the simulation display showing the picture representing the current state of the simulation is consistent with the rule's before-picture, this picture is replaced by the after-picture leading to the new state of the simulation.

These rules are relatively easy to use because they do not use many non-domain concepts. Furthermore, the mapping distance from the before- and after-pictures in the rule to the actual simulation is small because it uses the same, mostly visual, representation. Translating from a textual to a visual visual and back is not necessary.

Nevertheless, systems using visual rules still have problems. The goal of Cartoonist is to solve some of these problems by increasing the reasoning power of the underlying programming system without also increasing the complexity of the language. The four key problems are:

**Composing complex behavior from separate descriptions.** Each rule specifies a certain action of an object. Combining the behavior described by several rules is possible only by adding another rule. In addition to the objects avoiding each other, one might want to add that they also move in a certain way and still avoid each other.

**Dealing with temporal concepts.** Visual rewrite rules require additional representations to describe temporal concepts, for instance, to describe that an object keeps moving in the same direction. In a simulation, temporal concepts are important, and having to introduce many additional visual representations makes the rules more difficult to write and understand.

**Specifying a wide range of parallel behaviors.** Different problems require different treatments of which actions are executed in parallel. In rule-based systems, it is difficult to specify which of the treatments is the appropriate one.

**Describing behavior by specifying what should *not* happen.** Rules *generate* the next state of the simulation, which makes it difficult to describe a behavior by specifying that an object should avoid certain relationships to other objects. For instance, it is much

easier to say that an object should not end up next to another object than to specify what should happen in all the possible cases where one object is in the vicinity of another object.

Cartoonist aims to solve these problems while *retaining the advantages of the rule-based systems*. The end user should not be required to deal with additional non-domain concepts. Unfortunately, additional functionality often requires additional complexity at the user level. The additional functionality may well increase the complexity of the algorithms used by the programming environment, but this complexity must not be reflected by the end-user environment.

## 1.2 Context and Related Work

The Cartoonist framework involves many areas related to programming languages in general and, to some degree, artificial intelligence. Some of these issues are briefly discussed in this section and the connections between them and Cartoonist will be mentioned throughout the thesis. The last chapter will summarize how the Cartoonist framework relates to all of these areas.

### 1.2.1 Rule-Based Languages

This discussion is restricted to *forward-chaining* rule systems, or production systems [35]. Prolog, a backward-chaining rule system, is discussed in Section 1.2.2.1.

A production system consists of a working memory (WM), which represents the current state of the system, and the productions or rules, which modify the WM. The WM is normally represented as a set of facts and can be modified by adding and removing facts.

A rule is a condition/action pair of the form  $c \rightarrow a$ . The condition  $c$  is a pattern that is matched against the WM to test whether the rule is applicable. A rule with a matching condition may be applied and is assigned to the *conflict resolution set*. The *conflict resolution strategy* selects which of the rules in the conflict resolution set is applied to the WM. A rule is applied to the WM by executing the rule's action, which normally results in additions to and deletions from the WM.

The most prominent production system is OPS5; its advantage is its speed due to the incremental pattern-matching algorithm based on a Rete network [26]. Since the introduction of the Rete algorithm, faster algorithms have been developed [53]; however, the pattern and action languages stayed more or less the same as those introduced by OPS5 [12].

Most of the rule-based systems test only the current state to check whether a rule applies. Matching against a past state requires the user to explicitly store facts to access them later.



For instance, the direction in which an object moves cannot be inferred from the current state alone unless the direction or the object's previous position is also accessible in the current state.

An important part of a rule-based system is the choice of the correct conflict-resolution strategy, which decides which of the applicable rules is actually used to change the working memory. In OPS5 the user can choose from a small set of strategies. In Soar the conflict resolution strategy is implemented by the user with *preferences* [45, 59], and some systems use meta-rules to specify the conflict-resolution strategy [25, 36, 74].

In Cartoonist, the description of the simulation implements the conflict resolution strategy, however, the user does not know that. In Cartoonist, contrary to all the other rule-based systems with a programmable conflict resolution strategy, the specification of the strategy is completely in the problem domain. It is not at a meta-level, nor is it a domain independent built-in mechanism. This results in a flexible conflict resolution strategy that is adaptable to the problem domain.

## 1.2.2 Declarative Languages

Programming languages can be separated into *procedural* and *declarative* languages. Whereas production systems languages as discussed above are clearly procedural, programming languages based on logic and constraints tend to be declarative because they do not specify *how* something is computed, only *what* is computed. Burnett and Ambler stress that declarative programming will be of great importance in visual programming [14].

Cartoonist combines some important advantages of declarative and rule-based programming languages. Declarative programming concentrates on defining the relationships of the problem. These relations can hold between the objects in a simulation, for instance, defining geometric or temporal relationships. The mapping of these relationships to the actual operations in the simulation are left to the reasoning engine. This reduces the error-prone and time-consuming task of programming the operations by the end user.

The preference of using a forward-chaining mechanism as in Cartoonist over a logic programming language like Prolog is due to the fact that a simulation moves from one state to the next. A more proof-oriented model of computation on which most logic programming languages are based does not match well the needs of describing a sequence of states of a simulation.

```

member(X|[X|L]).
member(X,[_|L]) :- member(X,L).

max(X,Y,X) :- X >= Y.
max(_,Y,Y).

```

**Figure 1.1:** These are the definitions of a few Prolog predicates. `member(X,L)` succeeds if `X` is a member in list `L` and `max(X,Y,Z)` instantiates `Z` with the maximum of `X` and `Y`.

### 1.2.2.1 Logic Programming

The most famous logic programming language is Prolog [19]. Quite a few predicates, for instance `member` (see Figure 1.1), can be implemented efficiently in a purely declarative way. However, most non-trivial Prolog programs take advantage of the fact that Prolog can be used in a procedural manner. For instance, the usual definition of the predicate `max` shown in Figure 1.1 is not completely declarative, because the second clause takes advantage of the failure of the first clause. The declarative version of the second clause would be `max(X,Y,Y) :- X <= Y`.

Whereas a logic programming system puts emphasis on *proving* a result, a production system *generates* the result. For instance, in Prolog a clause of the form `p :- q` means “`p` if `q`,” that is, `q` implies `p`. Therefore, to prove `p`, one has to prove `q`.

The proponents of logic programming claim that declarative programming languages are generally easier to learn and use than are procedural programming languages because the programmer only has to specify the problem instead of how it is to be solved [73].

Prolog has also been extended with some ideas from the data-driven production systems. These systems claim to have a declarative forward-chaining rule system [30, 78]. Their rules are more or less of the form `c -> a` where `c` is the condition and `a` is the action, which is of the same form as the action in a production. The condition is of the same form as `q` in `p :- q` and the action `a` consists of assertions and retractions using the Prolog predicates `assert` and `retract`. These two predicates add and delete facts as side effects and are clearly not declarative.

The mere use of Prolog in the condition of a procedural rule does not lead to a declarative rule. Contrary to the authors’ claims, these systems do not provide declarative forward-chaining; however, a Cartoonist system does. To the best of my knowledge, Cartoonist is the first truly declarative forward-chainer.

Thus, in *Cartoonist*, the user can concentrate on the relations between the objects in the simulations and does not have to program the operation sequences necessary to transform one simulation state into the next. Furthermore, the forward-chaining character of the cartoons used to specify the objects' behavior is more natural to describe a state sequence of in a simulation.

### 1.2.2.2 Constraint Programming

Constraint logic programming (CLP) languages are another class of declarative languages, which are based on constraints [76]. These languages can also be viewed as an extension of logic programming languages with additional, declarative constraints on the variables' domains. These languages are generally used to solve combinatorial problems [20]. CLP programs tend to look more declarative than Prolog programs; however, this can probably be attributed largely to the kinds of problems generally solved by the two kinds of languages.

Although the specification of a problem with constraints can be completely declarative, often a procedural component is necessary to get the right behavior or the preferred result [8]. Constraints can even be used in combination with imperative programming languages [50].

Constraint-based and visual programming can also be combined to specify OPS5-like rewrite-rules [60]. This approach provides a simple integration of visual and constraint-based programming.

### 1.2.2.3 Planning

Planning is a problem-solving technique that orders actions to achieve a certain goal [17, 67, 77]. A plan is a representation of a (partially) ordered sequence of actions. In planning, actions are executed by applying an operator to a situation. An operator consists of a precondition, a postcondition, and the actual action. An operator can be applied to a situation if the precondition of the operator is fulfilled. The new state is computed by executing the action in the current state, which may change not only the planner's internal state but also the environment of the planner.

Once the operators are specified, the problems can be stated in a purely declarative fashion by specifying the initial state and the goal to be achieved. Although using a planner is not normally referred to as programming, the planning approach is somewhat similar to the *Cartoonist* approach.

### 1.2.3 Object-Oriented Programming

One of the aspects of object-oriented programming is the association of the object's actions with the object itself. The object decides whether it can deal with a certain message sent to it, and if it can, what it will do. Whatever can be attributed to an object will be encapsulated in this object.

However, if several objects interact with each other, it is not always obvious what actions should be attributed to which object. Cartoonist approaches this problem with the introduction of cartoons suitable to describe interactions.

In rule-based visual programming languages, the objects in the simulations consist of the depiction(s) of the object and possibly some value slots [72]. The visual objects in Cartoonist have a richer semantics because they have actions attached; the objects “know” what actions they can and cannot execute.

### 1.2.4 Visual Programming Languages

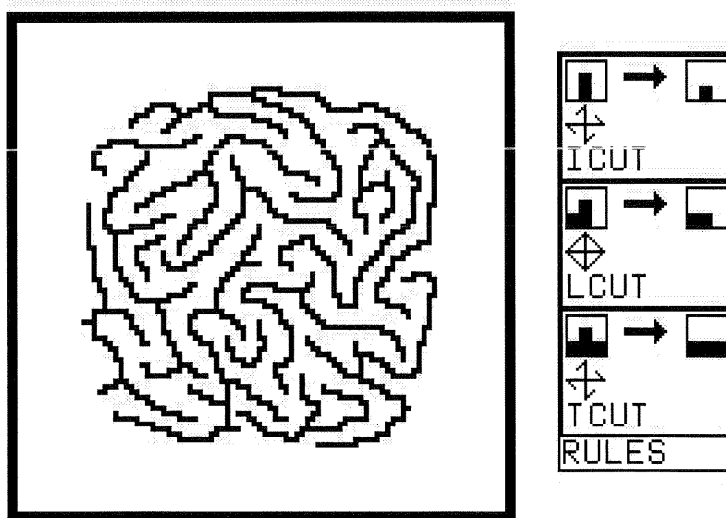
Visual programming describes a process by creating executable graphics built from spatially arranged pictorial objects. No completely satisfactory definition of visual programming languages exists; however, the one given here at least agrees more or less with those of Lakin [46] and Myers [56]. It is not so clear whether these languages should be called *programming* languages because often they are used to *specify* or *describe* a program, rather than to *implement* it.

In this dissertation, I concentrate on rule-based visual languages because they are well suited to describe graphical simulations. Other important inherently visual languages are *form-based* languages and *data-flow* languages [16, 32]. For a detailed classification of all the different visual programming languages and the related issues, see ref. [15].

There is evidence in favor of visual languages [2, 22], yet there are also results suggesting that textual languages are better than visual ones [61]. Neither a purely visual nor a purely textual language will be optimal for all problems, though. It is just as wrong to claim that visual languages are basically useless [11] as it is to justify visual languages simply because “a picture says a thousand words” or “graphical representations are more natural” [70].

In visual rule-based systems, the rules and the working memory are described with pictures. The simulation display makes visible some of the facts in the working memory. Sometimes certain auxiliary visual objects are hidden from the user, as in ChemTrains [5] and slot values; for instance, the age of a person is displayed only upon request by the user [51].

The rules consist of a before-picture and an after-picture. If the after-picture matches a part of the simulation display then the rule can be executed, or fired, by replacing the



**Figure 1.2:** A BitPict example. On the left is the simulation display and on the right are three rules reducing each bit string in the simulation to one pixel.

matched part in the simulation display with the after-picture (see Figure 1.2 for an example from [29]). The objects in the pictures range from pixels in BitPict [28] to icons in Agentsheets [66] and KidSim [72] to any picture drawn by the user in ChemTrains [6].

These systems describe graphical simulations. PROGRES, a visual graph-rewriting system [69], has a different goal. The visual elements in the simulation display and in the rules are restricted to graphs that can be used in a wide range of areas as description language [52]. For instance, in PROGRES a set of visual graph-rewrite rules can recognize all of the control flow diagrams, displayed in the simulation display, that correspond to goto-less programs.

### 1.2.5 Programming by Example

Programming by example [23] has a long history related to the field of *automatic programming* [4, 7]. As the name says, programs are created using examples as input.

KidSim uses programming by example to create the visual rules [72]. The simulation display is used to select an example for the before-picture in the rule. This picture is then modified, which results in the after-picture. This is a simple way of programming by example, yet this simplicity also reduces the user's possible misinterpretations of the meaning of the rule as defined by the programming system.

Kurlander's system, Chimera, takes multiple snapshots of a graphical system and infers the constraints between the objects in the snapshots [43, 44]. The inferred constraints can

then be used as a constraint-based program. A similar approach is taken in PURSUIT, a visual shell for *programming by demonstration* [55]. A program in script-form is inferred from a series of snapshots. While the program, based on the comic-strip metaphor, is being constructed, it is displayed in a visual state-based form. Because the visual state-descriptions can be edited, the creation of the program does not rely on the snapshots alone but allows human intervention, contrary to Frank and Foley's system [27], which relies completely on a domain-independent inference engine.

In Cartoonist, a simulation is described similar to that in KidSim. Cartoonist therefore uses, to some degree, example-based programming. However, it does not infer a program in a different language.

### 1.2.6 Behavior-Based Programming

The classic view of artificial intelligence is based upon reasoning on explicitly represented knowledge [9]. An intelligent agent is rational under the constraint of its resource limitations; that is, it always chooses the best actions given its knowledge and reasoning power [58, 71]. In the recent past, a more *situated* approach has been proposed in which the agent relies much more on the world as the representation to which the agent reacts. Brooks [10] has proposed a *behavior-based* approach, which resulted in the *subsumption architecture*.

A behavior-based architecture distinguishes itself from the representationalist architecture in that it is composed not of functional units, but of behavioral units. Simple behavioral units, or behaviors, can be composed resulting in complex behaviors. A PacMan-like game can be implemented relatively easily in a behavior-based framework [1].

The Cartoonist framework is behavior based, because the basic elements to describe a simulation are behaviors and not an explicit representation of knowledge and a reasoning engine. Refining the behavior of the objects differs from rule-based systems because the different descriptions can be combined dynamically in a subsumption-like architecture as discussed further below.

## 1.3 Organization of the Thesis

Chapter 2 motivates the use of cartoons as a mechanism to describe simulations. It introduces formally the concept of a cartoon without the more general framework based on temporal constraints. The preference scheme and the subsumption-like architecture are described and illustrated with a few examples.

Chapter 3 introduces the main ideas of *Cartoonist* in a demo-like fashion, providing a relatively informal account of what is a cartoon, how it differs from a rule, how it is used by an end user, and how and why it works.

Chapter 4 develops a framework in which existing sequential and parallel rule-based systems can be described and compared. *Cartoonist* is compared to *Agentsheets*, *BitPict*, *KidSim*, *ScienceShows*, and some non-visual rule-based systems. In an important sense, *Cartoonist* is more powerful than these systems. *Cartoonist* is implemented as an extension of a rule-based system based on a temporal extension of a Rete-network. The analysis of the system's time complexity shows that the current prototype tends to be slower than conventional parallel rule-based systems, which is due partially to the greater functionality of *Cartoonist*.

Chapter 5 describes how the prototype of *Cartoonist* is built on top of *Agentsheets*, providing yet another way to program in *Agentsheets*.

Finally, Chapter 6 summarizes how *Cartoonist* relates to the different areas mentioned in Section 1.2. Then the chapter discusses the contributions of this thesis work and suggests some possibilities for future work on *Cartoonist*.

In Appendix A, the general framework of temporal constraints is introduced. It suggests possible extensions of cartoons, which are shown to be special case of temporal constraints.

# Chapter 2

## Cartoons

If one is interested in describing visual simulations, it seems to be quite reasonable to start with animated cartoons, especially considering that the animations are described with drawings that are visual languages. This chapter discusses the comic-strip metaphor on which the concept of a cartoon is based. The metaphor suggests separating what the characters in the animations can do from what they should do. Next, visible and invisible states are introduced. Then the chapter defines the concept of a cartoon and how the voting scheme implements the subsumption-like combination of cartoons. Then, the Cartoonist engine is described and the voting scheme is illustrated. Finally, characteristics of cartoons are discussed. The more general framework of a cartoon as a special case of a temporal constraint is introduced in Appendix A.

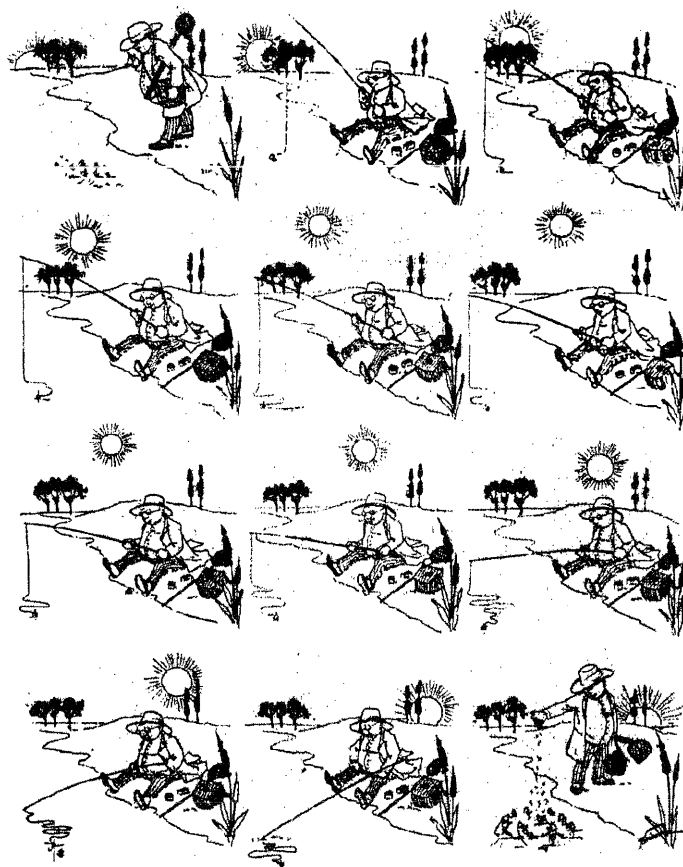
### 2.1 Comic-Strip Metaphor

A simulation can be described to another person in several ways. The most obvious way is to show the simulation itself, for instance as a movie. This description has the disadvantage that it is not necessarily clear *why* things happen in the simulation the way they do; that is, the explanation for the behavior may be missing.

Another possibility is a textual description. However, a textual description of a visual process forces the observer to map the textual concepts to the visual ones and vice versa, the latter at least while the program is debugged.

Before discussing how cartoons used in the Cartoonist framework are related to the comic-strip metaphor, we must clarify a few terms. The term *cartoon* is always used in a purely technical sense and never as a satirical drawing. A *comic strip* is a group of satirical drawings and looks quite similar to a (Cartoonist) cartoon; however, the “comic” might be a bit misleading. The reason for calling the concept a cartoon comes from the definition of



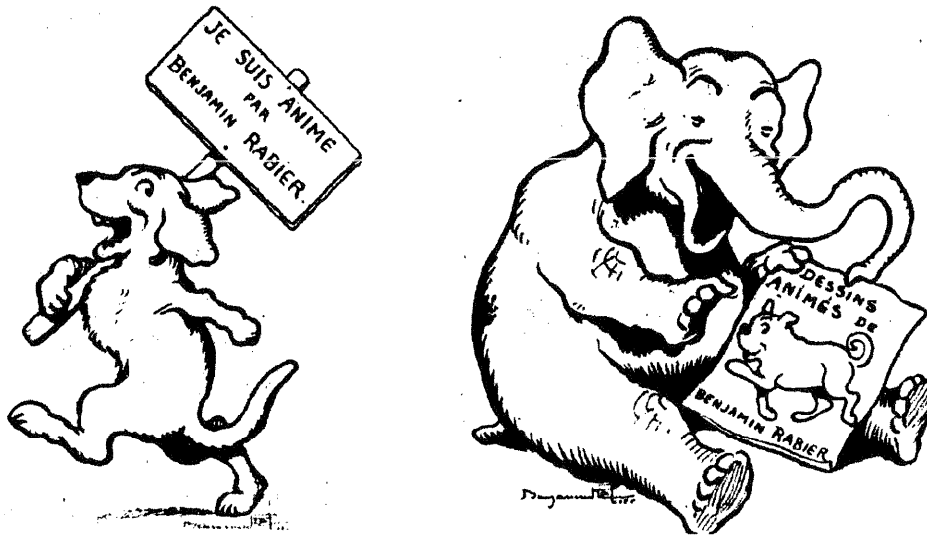


**Figure 2.1:** Théophile-Alexandre Steinlen: “Puisqu’ils ne veulent pas se laisser manger, ‘suicidons-les!’ ” *Le Chat Noir*, June 28, 1884.

an *animated cartoon*, which is “a motion picture made from a series of drawings simulating motion by means of slight progressive changes” [79]. There is one important difference. Cartoons are used to describe a *simulation*, which is quite different from an animated cartoon or an *animation*. An animation consists of basically one sequence; the movie always starts with the same scene and has the same ending. This is not true for a simulation, which normally is run with different initial states, and several simulation runs can be very different from each other.

Comic strips can visually describe a dynamic visual process. They can be seen as sequences of snapshots of the actual processes. A comic strip is similar to a movie, except that the latter, we are made to believe, is not a sequence of snapshots but a continuous flow of changes. Actually, comic strips were called the “movie theater for the poor” [39].

A comic strip such as the one in Figure 2.1 [21] tells one story only, that is, the initial situation and all the other situations will always be the exactly the same if the comic strip is animated or “run.” This is what differentiates it from a simulation run. The behavior of



**Figure 2.2:** Benjamin Rabier: Advertisement for the “Flambeau” series, an animated cartoon series released in 1917.

each of the characters in the comic strip—the fisherman, fishing pole, sun, and so on—could be told separately. Many different stories could be told, each starting with a different initial situation.

## 2.2 Characters

In Cartoonist a simulation described by cartoons contains the *characters* of the story. This section discusses the relationship among the characters, their actions, and the cartoons.

**CHARACTER:** A character is an object in a simulation that can execute a set of actions, for instance, move around, change its appearance, or make a sound.

A comic strip is a sequence of drawings of characters that can behave in certain ways. If the reader of the comic strip did not know the expected set of actions that could be attributed to a certain character, it would lose its meaning, as Figure 2.2 [21] illustrates.

A character is not just an icon but an object that has actions associated with it. These actions define the semantics of the character, which the character’s depiction does not. Instead of adding actions, the icons can be given some semantic content [65].

What actions a character *can* execute must therefore directly be associated with the character, that is, a character and its actions cannot be separated. The end user starts

with a set of characters, each with its own set of possible actions, and then describes the simulation by telling stories about the characters with cartoons.

Because actions are a part of a character, a cartoon does not describe an action. A cartoon, by specifying snapshots of the simulation, *coordinates* the actions of the characters but *does not generate* them. Each snapshot in a cartoon can be viewed as a constraint on what states may follow each other. Since the order of the constraints is important, a cartoon can be viewed as a temporal constraint.

### 2.2.1 Rules: Conditional Actions

Implementing simulations with conditional actions only can lead to problems related to the organization of the rules and description of interactions among several characters in the simulation.

Each rule describing a conditional action implements a potentially new action. The larger the number of rules the more unclear become the actions of the different characters in a simulation. If the user wants to change what a character can do, many rules may have to be changed and it might be unclear which ones should be changed. For instance, if a ball that can move in four directions—up, down, left, and right—is to be changed such that it also can move diagonally, the changes in the rules can be difficult and cumbersome.

It is also not always clear what the conditional actions should be to get a certain behavior. Things can easily get complicated, especially if several characters are involved in an interaction. It would be nice to describe such an interaction without having to resort to the level of conditional actions. However, rules do not allow this.

### 2.2.2 Cartoons: Structuring Actions

Characters have a specified set of actions, so which actions a character can execute are always clear. For instance, a ball may move to any of the four directions (up, down, left, and right) but must not ever move to a place occupied by another object.

Cartoons are used to describe the characters' behaviors, not their actions. Cartoons describe what the characters should do, or more exactly, what state they should accomplish with one of their actions. Given the huge space of possible action sequences, cartoons can choose those sequences that are consistent with the given story.

Another advantage of separating what a character can do from what it should do is that the same story can be told in a situation where the characters can execute different actions. The same cartoons can be used but the actions are changed. For instance, in the PacMan

example in Chapter 3, changing from a situation in which ghosts can move in only four directions to one in which ghosts can also move along the diagonal was effortless.

## 2.3 Visible and Invisible States

A state in a simulation is an arrangement of the objects that participate in the simulation. States are internally represented as a set of facts and may be shown externally as a picture in the simulation display. A fact is of the form

$$\langle \text{fact name} \rangle (\langle \text{arg}_1 \rangle, \langle \text{arg}_2 \rangle, \dots, \langle \text{arg}_n \rangle),$$

where the fact name and all the arguments must be constants. For instance, the state

$$\{ \text{at}(\text{ball}, \text{b1}, 2, 3), \text{empty}(3, 3) \}$$

means that a ball with the name b1 is at the coordinate (2,3) and the square at coordinate (3,3) is empty.

Each state in a simulation is either *visible* or *invisible*. A visible state is shown in the simulation display, whereas an invisible state exists internally to the Cartoonist system, but are not shown in the simulation display. Invisible states make the system appear to execute actions in parallel as follows. Assume the system goes through the states  $s_1, s_2, \dots, s_k$ , where all states are invisible except for  $s_1$  and  $s_k$ . Each state  $s_i$  is generated by an action applied to the previous states  $s_{i-1}$ . The observer of the simulation sees only the transition from state  $s_1$  to  $s_k$ ; as a result the actions leading to the states  $s_2, \dots, s_k$  appear to be executed in parallel.

## 2.4 Cartoons

A cartoon is a sequence of constraints that describes a sequence of states. The constraints in a cartoon are represented by pictures which specify the required features of states.

A state sequence  $s_1, s_2, \dots, s_k, s_{k+1}, \dots, s_l$  is consistent with a cartoon  $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$  if

1. the states  $s_1, \dots, s_k$  are consistent with the constraints  $C_1, \dots, C_k$ , respectively,
2. state  $s_k$  is the current state shown in the simulation display,
3. state  $s_l$  is consistent with constraint  $C_{k+1}$ , and
4. the states  $s_1, \dots, s_k$  are visible and the states  $s_{k+1}, \dots, s_{l-1}$  are invisible.

A technical note: Constraints often contain variables; when they do, the same variable substitution must be used for all constraints  $C_1, C_2, \dots, C_{k+1}$ . The voter is then the cartoon instance  $C_1^\sigma \rightarrow C_2^\sigma \rightarrow \dots \rightarrow C_{k+1}^\sigma$ .

The first two parts of the above definition say that constraint  $C_k$  describes the current state and the constraints  $C_1, \dots, C_{k-1}$  describe the states in the past. The third part says that the last constraint  $C_{k+1}$  in the cartoon describes the last state  $s_l$  in the sequence and thus, parts of the next visible state. Since  $k \geq 1$ , there is at least one constraint describing the current state and a constraint describing a state  $s_l$  in the future. Zero or more constraints refer to past visible states of the simulation.

All states described by the constraints  $C_1, \dots, C_k$  must be visible, that is, they can be seen by the observer of the simulation. This part of the cartoon can be viewed as the equivalent of the condition in a rule, except that the cartoon can access states in the past whereas rules normally cannot. The sequence of invisible states following  $s_k$  will eventually lead to a visible state, which will be the next state shown in the simulation display.

### 2.4.1 Cartoons Are Not Rules

Cartoons are not just rules that can refer to the past in the condition or before-picture. Referring to the past could just as easily be done in rules, which still would leave the most important difference between rules and cartoons untouched.

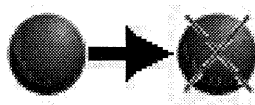
Rules *generate* a new state and cartoons *describe* a sequence of existing states. A cartoon never changes a state; however, it *recognizes sequences of states*. Therefore, a cartoon can be viewed as a subset of a finite automaton accepting certain state sequences. If it accepts a sequence, it votes for the last state in the accepted sequence.

As shown below, rewrite rules are used to implement the characters' actions. The characters execute these actions and generate possible future states. In summary, the cartoons select one of the state sequences generated by the rules.

In rule-based systems the user writes the rules to generate the state sequences, and the built-in conflict resolution strategy selects one sequence. In a Cartoonist system, the system generates the state sequences, and the user writes the cartoons that select one sequence.

## 2.5 Voting Scheme

The cartoons select the next state generated by the rules, as follows. Suppose the simulation has proceeded through states  $s_1 s_2 \dots s_{n-1}$  and is in state  $s_n$ . Using the rules, all possible next states are constructed, say  $s_{n+1}^1, \dots, s_{n+1}^w$ . The possible state sequences  $s_1 \dots s_n s_{n+1}^1,$



**Figure 2.3:** This cartoon has two constraints and describes that the ball is moving around randomly.

$s_1 \dots s_n s_{n+1}^2, \dots, s_1 \dots s_n s_{n+1}^w$  are then evaluated as follows. Each possible sequence may be consistent with some cartoons and not others. A cartoon that is consistent with a sequence is called a *voter* for the last state of this sequence. The sequence with the most powerful voters is chosen, and its last state is selected as the next state as the simulation.

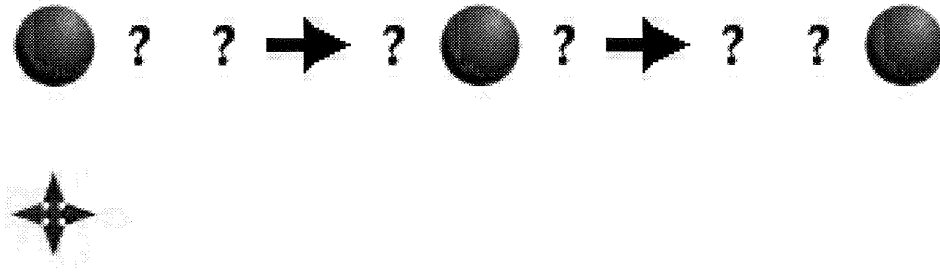
Every state that is selected as a new state is assumed to be invisible until in some invisible state  $s_i$  no state  $s_i^j$  is found for which there is a voter. This can happen because no new state could be generated by the rules, in which case the simulation halts. The other reason is that no cartoon is consistent with the state sequence. In this case,  $s_i$  is set to visible and shown in the simulation display; then, the simulation is continued.

### 2.5.1 Preferences between Cartoons

In comparing voters, some cartoons are given more weight than others. In a Cartoonist system the user can state that a certain cartoon  $\alpha$  is preferred over another cartoon  $\beta$ , written  $\alpha \succ \beta$ . This preference relation is transitive, that is, if  $\alpha \succ \beta$  and  $\beta \succ \gamma$  then  $\alpha \succ \gamma$ . If no preference relation is stated for two cartoons  $\alpha, \beta$  and none can be deduced using transitivity, then the two cartoons are indifferent. The preference relation is similar to the linear orderings of the rules used in rule-based systems except that  $\succ$  is partial ordering.

In addition, a heuristic is used to order cartoons in some cases when no preference between the two cartoons has been stated. A cartoon is preferred over another cartoon if it has more constraints. This does not change the preference relation stated by the user because it is only used if two cartoons are indifferent. However, it often leads to the expected behavior without the user having to state preferences. For instance, two of the cartoons used in the negative programming example in Chapter 3 are in Figures 2.3 and 2.4. The cartoon in Figure 2.3 has fewer constraints than the one in Figure 2.4. Since the user has not stated a preference between the two cartoons, the number of constraints is used as a tie-breaker; therefore, the cartoon in Figure 2.4 is preferred. Without this, the ball might move around more or less randomly, because the random-move cartoon has a better chance of producing a voter than the cartoon with three constraints.

Thus, the heuristic often reduces the number of preferences that have to be explicitly stated by the user. However, the functionality of cartoons does not change.



**Figure 2.4:** The cartoon has three constraints and describes a ball moving in the same direction.

The preference relation and the heuristic were described for cartoons. They directly translate to voters in the sense that a voter  $v_1$  resulting from cartoon  $\alpha_1$  is preferred over a voter  $v_2$  generated from cartoon  $\alpha_2$ , if  $\alpha_1$  is preferred over  $\alpha_2$ , that is,  $v_1 \succ v_2$  if  $\alpha_1 \succ \alpha_2$ .

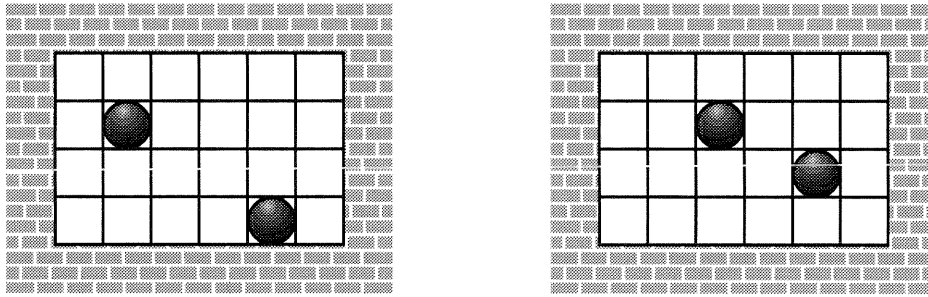
### 2.5.2 Comparing Sets of Voters

A state is selected if it has the best set of voters. This makes it necessary to define how sets of voters are compared. The idea is that the preferences stated have to be followed; that is, if a voter  $v$  votes in favor of state  $s_1$  and  $v$  is preferred over all voters of all the other states, then  $s_1$  is chosen independently of any other considerations.

Let  $V_i$  be the set of voters voting for state  $s_i$ .  $V_i \succ V_j$  means that the set  $V_i$  of voters is preferred over the set  $V_j$ . The best states are those for which there is no state with a better set of voters, that is,  $s_i$  is a best state if  $\neg \exists j. V_j \succ V_i$ .

The algorithm to test if  $V_i \succ V_j$  is true is given below. It uses two rules: Indifferent voters neutralize each other, and once a voter is found that is better than any voter in the other set, a decision can be made immediately.

1. if  $V_i$  is empty then return *false*
2. if  $U_i$  is empty then return *true*
3. let  $v_i$  and  $v_j$  be the best voters in  $V_i$  and  $V_j$ , respectively
  - (a) if  $v_i \succ v_j$  then return *true*
  - (b) if  $v_j \succ v_i$  then return *false*
  - (c) remove  $v_i$  from  $V_i$  and remove  $v_j$  from  $V_j$  and then go to 1



**Figure 2.5:** The situation on the left precedes the situation on the right, where the upper ball moved right and the lower ball moved up. The grid is useful for the discussion, but is not drawn in the actual simulation.

### 2.5.3 Subsumption-Like Architecture

This voting scheme provides a natural implementation of subsumption-like specifications of combinations of behaviors, as seen in the Pachinko and the PacMan examples. By adding cartoons to a simple specification one ensures that behaviors will satisfy several layers of constraints, if possible, but will satisfy more important constraints if constraints conflict.

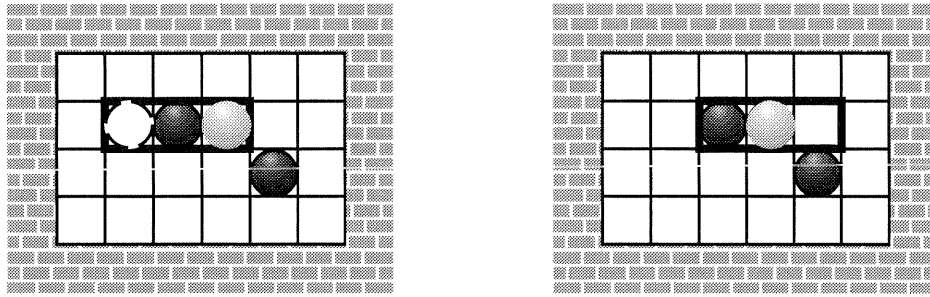
## 2.6 Illustration of the Voting Scheme

This section explains how the Cartoonist engine works. The matching process, the interaction between rules and cartoons, and the voting mechanism are explained. Formalisms are postponed to the following chapters whenever possible. To visualize the mechanisms better, the pictures are not snapshots from the ASC prototype but created with a drawing program.

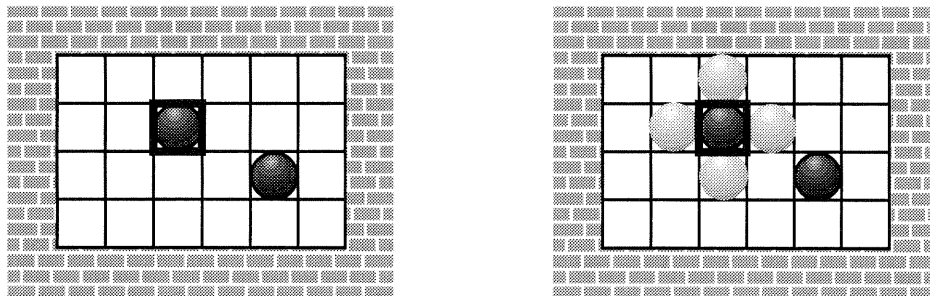
A *character* is an object in the simulation consisting of an icon and a set of actions. The icon determines the character's appearance. The set of actions determines what can be executed by the character, for instance, to move around or to play a tune. A wall has no action and a ball can move east,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

In Figure 2.5, two balls are surrounded by a wall and therefore can not leave the 24 squares. The ball on the left moved one square to the right and the ball at the bottom moved one square up. The rest of the chapter discusses where the ball at the top left is moving and why. I will refer to the state on the right in Figure 2.5 as the current state with the past shown in the left part of the figure.





**Figure 2.6:** The left picture shows a voter with the previous ball position (left) and the potential future position (right). The right picture shows a constraint instance that is not a voter. The potential future ball position is also shown.



**Figure 2.7:** The left picture shows where the cartoon is matching and the right picture shows all the potential future ball positions.

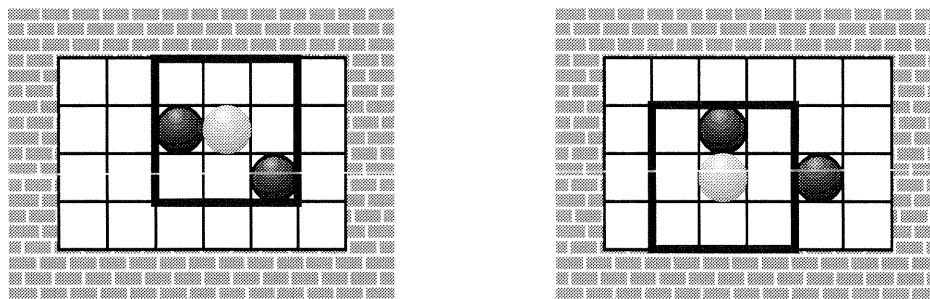
### 2.6.1 Voters

An instance of a cartoon describes a certain part of a state and its history. The voter is a specific cartoon instance whose last constraint is consistent with a state in the immediate future.

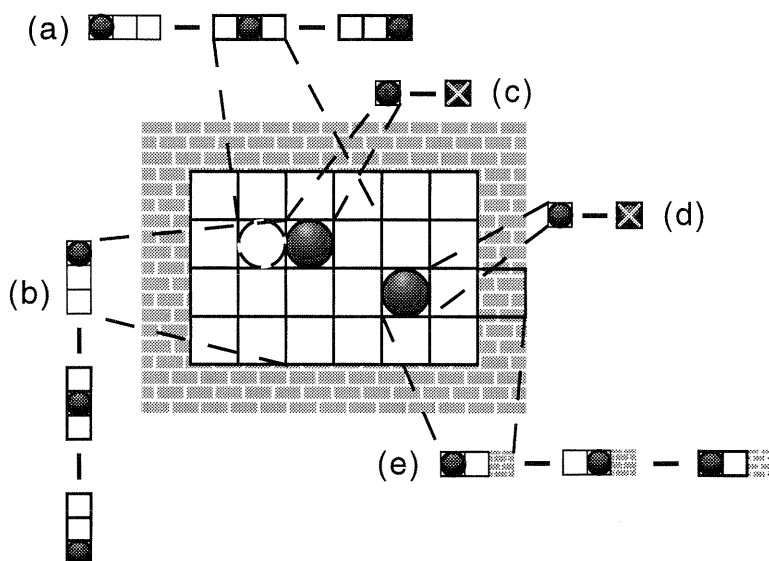
Figure 2.6 shows the current state of the simulation and how the cartoon  $\blacksquare \square \rightarrow \square \blacksquare \rightarrow \square \blacksquare$  is matched against the state. In the left picture, the cartoon matches the former (empty ball), current (dark gray ball), and potentially future position (light gray ball) of the ball. This voter directs the ball to the right. The right picture shows a cartoon instance not voting for the ball going to the right, because the instance is not a voter.

Figure 2.7 shows on the left how the cartoon  $\blacksquare \rightarrow \blacksquare$  is matched against the ball. It is also matched similarly against the other ball. In the picture on the right, the potential positions of the ball in the same figure are shown. Matching the cartoon as in the left picture leads to four voters, each voting for another action leading to the four potential locations (shown as light gray balls) in the right picture.

Figure 2.8 shows how the cartoon  $\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix} \rightarrow \begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$  is matched against the top left ball. The picture on the left shows that the cartoon cannot vote in favor of the ball going to the right



**Figure 2.8:** Two of the four ranges that have to be checked for the top left ball and the avoid-cartoon are marked. Moving to the right does not get a vote from this cartoon, as the left picture shows. The avoid cartoon voted in favor of the ball moving down (and left and up).



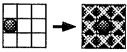

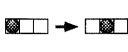

**Figure 2.9:** Some of the matches are shown. For instance, the cartoon instance (a) is a voter whereas (b) is not. See the text for a more detailed explanation.

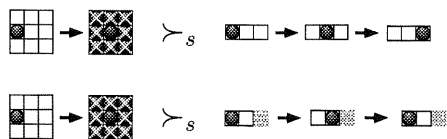
because of the ball at the bottom right. However, the picture on the right shows a match that leads to a vote for the ball going down one square. Going up and to the left will also get a vote from this cartoon. Because all the constraints of a cartoon must be used to be eligible to vote, and the last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

Figure 2.9 shows a few voters and cartoon instances. Instance (a) is a voter because its second-to-last constraint describes the current state. It votes for the ball moving to the right. Instances (b) and (e) are no voters. Finally, instances (c) and (d) vote for four actions each, namely, moving either ball in any of the four directions.

### 2.6.2 Voting Scheme

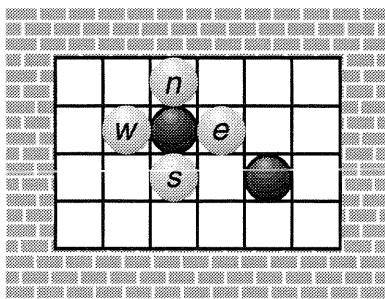
Figure 2.10 illustrates the voting scheme that implements the subsumption-like architecture in which the behaviors here described with cartoons are organized in a hierarchy according to the importance of the behaviors. The most important cartoon that applies to the current situation is chosen. If this behavior is ambiguous, that is, if several possible actions are consistent with the behavior, then less important behaviors consistent with the more important behavior can be chosen to break ties between the actions, and so on. For instance, 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.

The cartoons are organized in the subsumption hierarchy with the static preference relation between the cartoons defined by the user. A cartoon  $\alpha_i$  being preferred over  $\alpha_j$  is written as  $\alpha_i \succ \alpha_j$ . (See Section 2.5.1 for the definition of the  $\succ$  symbol.) The preference relation is partial and it has to be stated only for those cartoon pairs for which an ordering is necessary. As mentioned earlier, the static preferences between the cartoons are as follows. The static preference between the random-walk cartoon   $\rightarrow$   and the cartoon   $\rightarrow$   is omitted, because it is not necessary since it applies for any ball movement.



The action that the ball at the top in Figure 2.5 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 2.10). The columns of the table are named with the actions and the rows with the cartoons. Action  $e$  gets two votes from the voters generated by the cartoons on lines (a) and (c) and the other three actions get two votes each from the voters generated by the cartoons in rows (a) and (b). These sets of voters have to be compared according preferences between the voters. Let  $v_i$  and  $v_j$  be instances of cartoons  $\alpha_i$  and  $\alpha_j$ , respectively. The definition of the preference relation between the voters then basically says that a voter  $v_i$  is preferred over a voter  $v_j$ ,  $v_i \succ v_j$ , if

1.  $\alpha_i$  is preferred over  $\alpha_j$ , that is,  $\alpha_i \succ \alpha_j$ , or



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

**Figure 2.10:** The table at the bottom shows what votes the cartoons generate for the actions moving east, west, north, and south. The letters in the table stand for “voter,” “cartoon instance,” and an x stands for no instance at all. See the text for a more detailed discussion.

2.  $\alpha_i$  and  $\alpha_j$  are indifferent and  $v_i$  is older than  $v_j$ , that is,  $v_i$  has more of a history than  $v_j$ .

Since the latter condition is true if  $\alpha_i$  has more constraints than  $\alpha_j$ , the preference can be computed when the cartoon is computed. It is not necessary to delay until run time.

The action with the best set of votes, that is, the action with the best support, will be executed, where the sets of votes for the actions are

$$\begin{aligned}
 e: & \{ a, c \} \\
 w: & \{ a, b \} \\
 n: & \{ a, b \} \\
 s: & \{ a, b \}
 \end{aligned}$$

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

$$\begin{aligned} e: & \{ c \} \\ w: & \{ b \} \\ n: & \{ b \} \\ s: & \{ b \} \end{aligned}$$

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

$$\begin{aligned} w: & \{ \} \\ n: & \{ \} \\ s: & \{ \} \end{aligned}$$

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

## 2.7 Characteristics of Cartoons

This section discusses the characteristics of cartoons, most of which cannot be found in rule-based systems. The main reason for the different characteristics of rules and cartoons is that rules *generate* states and cartoons *recognize* sequences of states. Using cartoons to describe a process, the user can avoid concern about how the states are generated, that is, what actions are executed. This can be powerful for declarative programming and using emergent or high-level relations.

### 2.7.1 Referring to Past and Present in the Condition

Recall that a cartoon is of the form  $C_1 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$ , where  $k \geq 1$ , and  $C_k$  describes part of the current state. The constraints  $C_1, \dots, C_{k-1}$ , if any, refer to the past. If  $k = 1$  the cartoon is of the form  $C_1 \rightarrow C_2$  and is conditional on the present state only. Referring to a past state is the same as referring to the present state or referring to a future state. A cartoon describes a sequence of states and is synchronized with the states so that the current state is consistent with the second to last constraint.

### 2.7.2 Using Emergent Relations to Describe the Future

Each relation is either a high-level relation or a primitive relation. A high-level or emergent relation can be defined as a relation that cannot appear in the right-hand side of a rule.

Although a high-level relation cannot be easily generated, it is generally simple to *recognize* it. For instance, point  $x$  is closer to  $y$  than to  $x$  is easy to recognize, however, it is

not clear where to put the points because there are so many possibilities. Distance can be a high-level relation, as the PacMan example in Chapter 3 showed.

The simplest kind of cartoon with two constraints  $C_1 \rightarrow C_2$  can show the special status of emergent relations in cartoons. This cartoon does not have the same functionality as a rewrite rule and does not say anything about *how* the current state described by  $C_1$  is changed into the next state described by  $C_2$ . Therefore,  $C_2$ , the cartoon's right-hand side, does not need to be restricted to primitive geometric relations as a rule's right-hand side is.

Any relation that can be *recognized*, no matter how complex it—in principle it could be a predicate proven by a Prolog program—can be used, not only in the condition of the cartoon but also in the constraint  $C_2$  describing the future state. This is important, because people are good at diagrammatic reasoning due to their ability to recognize and use emergent relations [47, 40]. They should not be restricted to primitive geometric relations.

The PacMan example in Chapter 3 uses a cartoon of the form

$$\left\{ \begin{array}{l} \text{the distance between ghost and PacMan is } d \\ \text{the distance between ghost and PacMan is smaller than } d \end{array} \right\} \rightarrow \text{to describe that ghosts}$$

track the PacMan. The distance between the two characters in the simulation can easily be recognized; however, it would not make sense to use the distance relation in the same way in the right-hand side of a rule.

High-level relations can be used in the description of any state, past, present, or future. For instance, any formula in first-order logic can be used as long as it is used as a high-level relation. Negation has especially proven to be of great value, as a few examples in Chapter 3 showed.

### 2.7.3 Declarative Forward-Chaining

Forward-chaining rule-based systems provide a natural mechanism to describe a sequence of states. However, the action of the rules are restricted to primitive relations which often makes difficult to describe the next state succinctly. Cartoons are not restricted by this constraint.

If the state constraints consist of primitive geometric relations only, that is, no high-level relations are used, then the cartoon  $C_1 \rightarrow C_2$  looks like a rewrite rule. However,  $C_1 \rightarrow C_2$  is still different: A cartoon does not describe *how* the future state is generated. For instance, the cartoon

$$\{\text{rook}(r), \text{at}(r, \text{h1})\} \rightarrow \{\text{at}(r, \text{f1})\}$$

describes, in the domain of chess, that the rook  $r$  moves from h1 to f1. This move can be accomplished either by simply moving the rook from h1 to f1 or by castling to the king side. Two rewrite rules are needed to express the two possible moves as follows.

$$\begin{aligned} & \{\text{rook}(r), \text{at}(r, \text{h1})\} \rightarrow \{\text{rook}(r), \text{at}(r, \text{f1})\} \\ & \{\text{king}(k), \text{at}(k, \text{e1}), \text{rook}(r), \text{at}(r, \text{h1})\} \rightarrow \{\text{king}(k), \text{at}(k, \text{g1}), \text{rook}(r), \text{at}(r, \text{f1})\} \end{aligned}$$

Although declarative forward-chaining mechanisms exist, they are not more declarative than “normal” rewrite rules. Gaspari [30] and Wallis and Moss [78] extend Prolog with data driven rules and call them declarative forward-chaining rules. What such a rule amounts to, though, is a rule of the form  $l \rightarrow r$  where  $l$  is a Prolog expression and  $r$  is an assertion or retraction of a Prolog fact—nothing but a procedural rule with a declarative condition. Due to the explicit use of assertion and retraction, these languages are even less declarative than Prolog itself. Thus, cartoons are, to the best of my knowledge, the first real declarative forward-chaining “rules.”

#### 2.7.4 Partial Specification of Actions

A cartoon describes only what state has to be reached, not how this is accomplished. Therefore, more than one action can change the current state satisfactorily, as the chess example showed. This characteristic of cartoons allows the user to specify just what is necessary, not more. It simplifies descriptions, because special cases, the trouble-makers of programming, can often be ignored.

This partial, or ambiguous, specification of actions allows the user to refine the behavior incrementally. Several cartoons can be combined to reduce the set of possible actions, as described in Appendix A. Assume that cartoon  $\alpha_1$  chooses set  $A_1$  of actions, while  $\alpha_2$  suggests the actions in set  $A_2$ . If  $A_1 \cap A_2 \neq \emptyset$ , then an action from  $A_1 \cap A_2$  is chosen.

#### 2.7.5 Flexible Description of Parallel Actions

In simulations, parallel actions should often be executed in parallel for semantical reasons, not for increased efficiency. It matters whether ice melts in parallel at the surface, or whether balls can roll down a slope without spreading out while rolling.

Cartoons simplify describing different kinds of parallelism, and the user does not even have to be aware of the parallelism. This is not true for current visual rule-based systems, which are restricted with respect to what kinds of parallel actions they can execute. Chapter 4 discusses these issues in detail.

# Chapter 3

## Preview of the Cartoonist System

This chapter introduces the Cartoonist framework in a demo-like fashion. First, some of the existing visual rule-based systems are described. After introducing some terminology and describing what a cartoon is, the Cartoonist prototype built on top of Agentsheets is used to present the end-user view of Cartoonist. Then, how the Cartoonist engine works, something the end user does not need to know, is explained.

### 3.1 Rule-Based Visual Programming Systems

This section introduces visual rewrite rules and describes some of the rule-based systems to which Cartoonist will be compared throughout this dissertation. These systems are compared to Cartoonist in detail in Chapter 4 using a general framework for sequential and parallel rule-based systems.

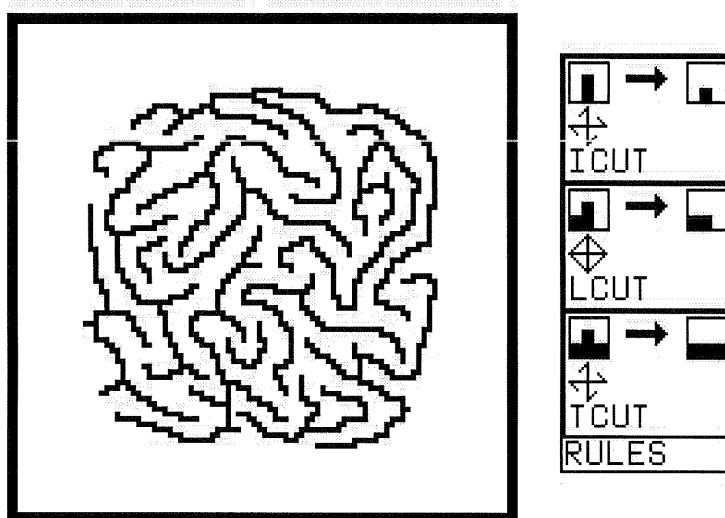
#### 3.1.1 Rules

In rule-based systems, rules are used to change the state of the system. In the visual systems considered in this section, the system state is usually represented to a large degree by the depiction of the simulation, also called the simulation display.

A visual rewrite-rule consists of a before-picture and an after-picture. A rule may be applied, if the before-picture describes a part of the simulation display, that is, if the simulation display *matches* the before-picture. A rule is applied, or fired, by replacing the before-picture with the after-picture in the simulation display.

Often, more than one rule could be applied, but only one is chosen by the *conflict resolution strategy* and then fired. Visual rule-based systems tend to use a simple, linear ordering of the rules. If the before-pictures of two rules match, then the rule is fired that appears earlier in the list. Some systems also allow the random choice of a subset of rules.





**Figure 3.1:** On the left is the simulation display of BitPict, and on the right are three rule manipulating patterns of pixels.

Rules *generate* a new state of the simulation by *changing* the simulation display—actually by redrawing a part of the display.

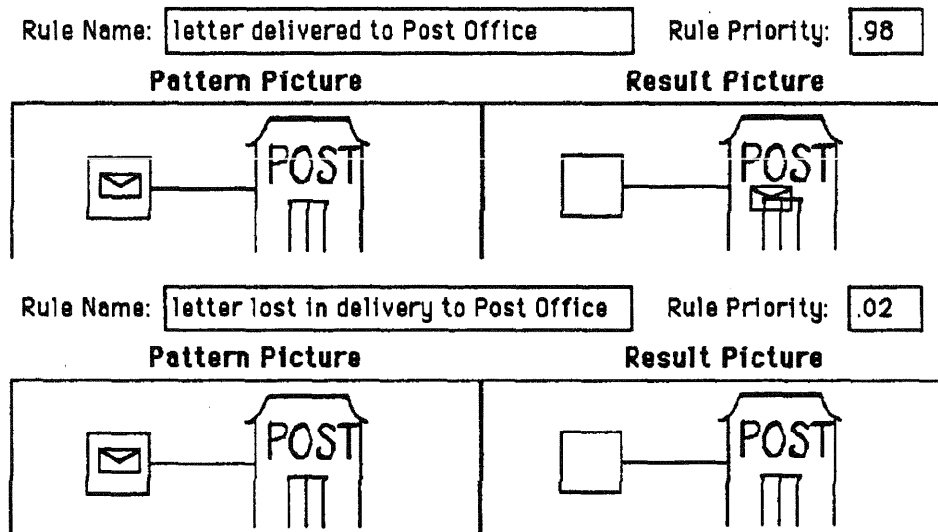
### 3.1.2 BitPict

BitPict works at the pixel level, that is, the before- and after-pictures test and modify patterns of pixels, as shown in Figure 3.1 [29]. The rules shown in the figure reduce a string by one pixel with each application until each string consists of one pixel only. They are ordered sequentially and the first that matches is fired. The markers below the before-pictures express that rules should be used also when the patterns are rotated or mirrored. The implementation of BitPict is simple and fast because the graphical objects and the patterns are of such a simple form. For the same reason, the system is not very expressive.

Rules can also be organized in groups that are linearly ordered. For instance, once the group of rules in Figure 3.1 has reduced the strings to a few pixels, no further rule in this group applies. Then, another group of rules is selected that counts the remaining pixels.

### 3.1.3 ChemTrains

The graphical objects in ChemTrains are much more complicated than the pixels in BitPict [6]. Any object drawn by the user can be used, and the main relations are *connectivity* and *containment*. Two objects can be connected with an arc, and an object can be completely



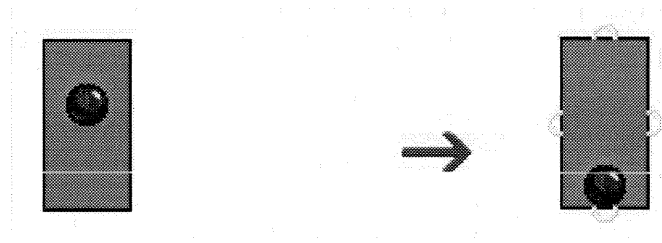
**Figure 3.2:** The rule at the top delivers the letter and is fired with a probability of 98%. The rule at the bottom loses the letter. These are parallel ChemTrains rules; therefore the letter is delivered with a probability of 98%.

contained in another object. ChemTrains is the only system described in this section that is not purely grid-based, although a grid can be implemented using connectivity and containment. Figure 3.2 shows two ChemTrains rules [5] in which the box on the left is connected to the post office and the letter is initially contained in the box. The graphical objects are a letter, a box, and a post office. The rules are ordered linearly; however, groups of rules can be at the same place in the list, meaning that one of these rules is chosen randomly if no higher prioritized rule already has fired. As the figure shows, these parallel rules can be associated with a probability (called “rule priority” in the picture).

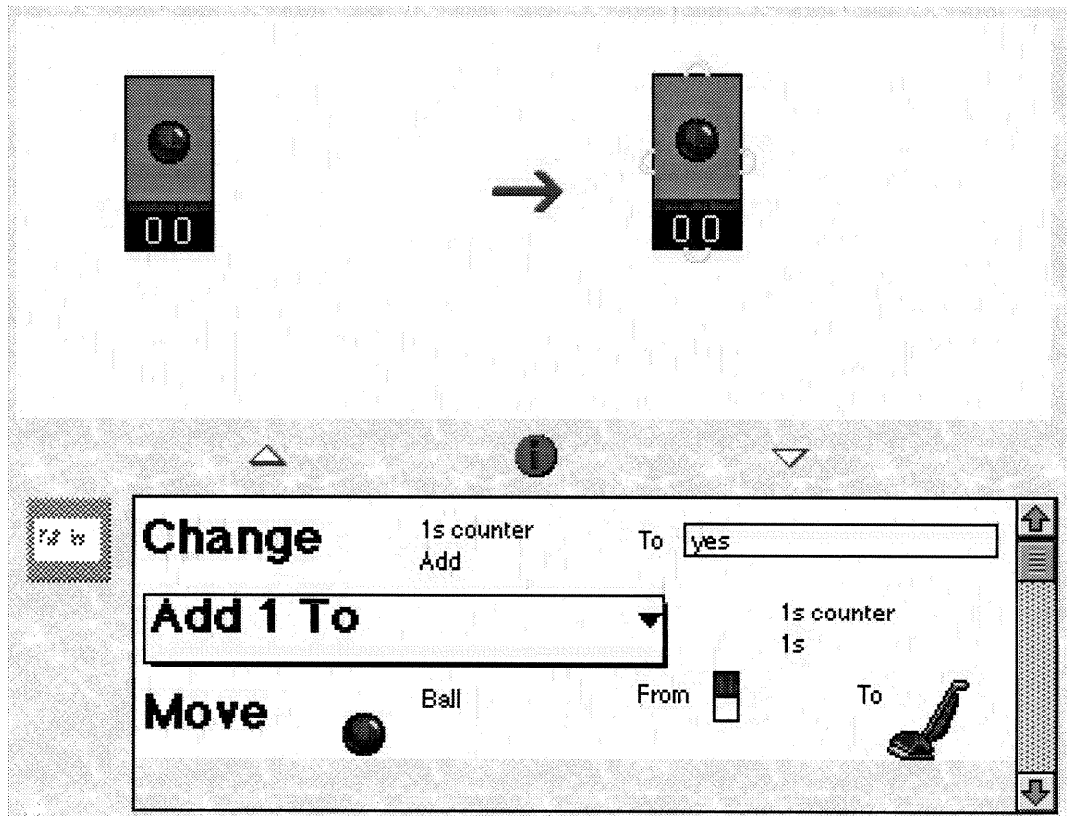
ChemTrains’ rules are powerful; for instance, a Turing Machine simulation requires just one rule. However, it is not easy to find such a rule, and understanding it can be difficult, too.

### 3.1.4 KidSim

KidSim is a visual programming language based on grids and icon-like graphical objects that are located in the squares of the grid [24, 72]. A rule describing a falling ball is shown in Figure 3.3. The graphical objects can be any size, but usually they fit in a cell of the grid. The rules are ordered linearly, and groups can also be ordered in parallel, similar to ChemTrains.

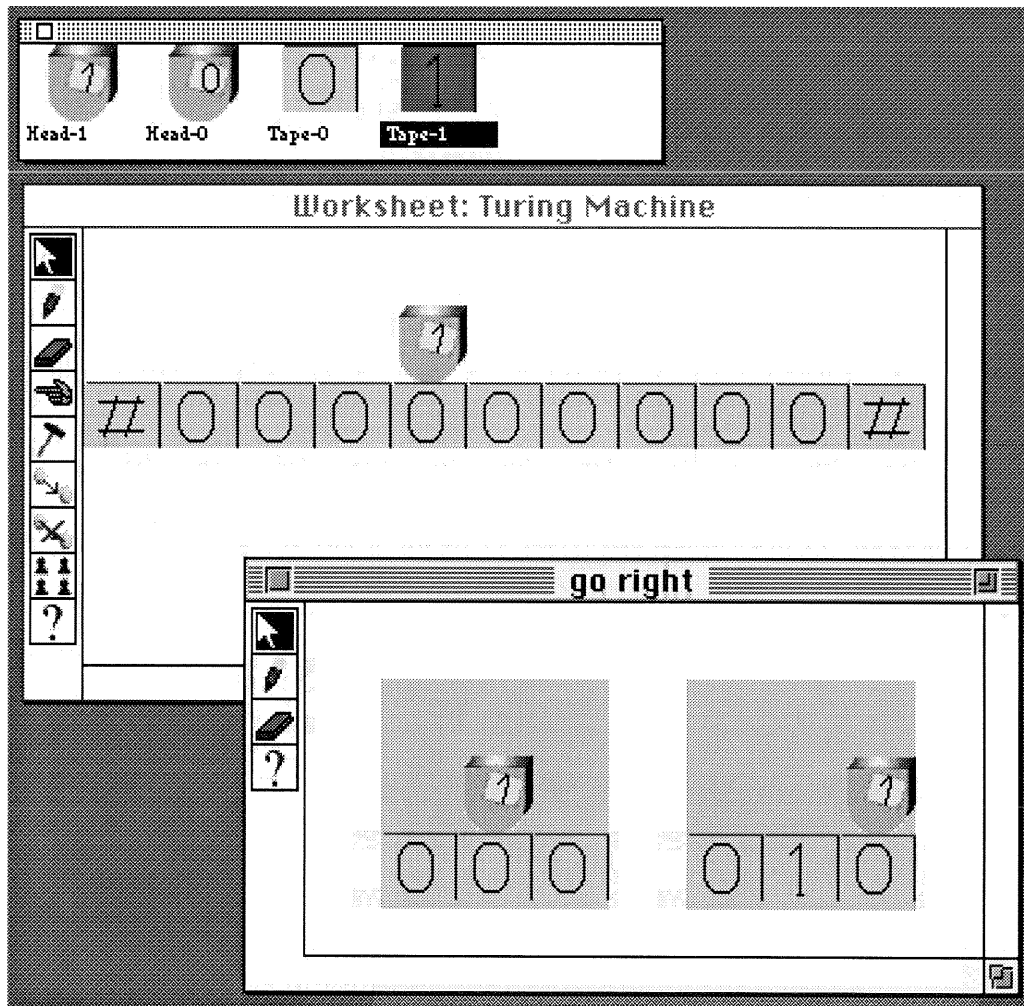


**Figure 3.3:** A rewrite rule in KidSim describing how a ball falls down one (invisible) square. The squares are made visible to the user during the drawing process of the rule.



**Figure 3.4:** This rule increments the “1s counter” slot. The visual part is used only to select the rule, not to change anything.

The rules are attached to the objects; for instance, the rule in Figure 3.3 is attached to the ball. An object can also have slots containing values that are invisible during the simulation run. The rule in Figure 3.4, for instance, increments the slot “1s counter.” This combination of visual and textual programming leads to great functionality; however, the textual component also complicates the process of programming in KidSim.



**Figure 3.5:** The Turing Machine programmed in Agentsheets. The rule describes the action that makes the head write a “1” and then move to the right.

### 3.1.5 Agentsheets

Agentsheets is a programming substrate to create domain-oriented programming and simulation environments [64, 66]. The visual representation of the graphical objects is similar to the one used by KidSim. The agents are in the cells of a grid, called the agentsheet. Normally, these agents are programmed using CommonLisp; however, the user can also take advantage of rewrite rules, as shown in Figure 3.5. Whenever Agentsheets is referred to, it is assumed by default that the the rewrite-rule extension of Agentsheets is meant.

At the top of Figure 3.5 is the palette with agents that can be used in the worksheet, which is Agentsheets’ simulation display. The rule is displayed in the window named “go right” where the before- and after-pictures are restricted to  $3 \times 3$  grids. This size is large

enough for many problems. The rules are also attached to the agents as in KidSim; for instance, the one in Figure 3.5 belongs to the gray head that is above the tape.

The rules are normally linearly ordered. However, Agentsheets also allows the implementation of more elaborate parallel execution models, as will be discussed in Chapter 4.

### 3.1.6 ScienceShows

ScienceShows is based on the same visual representation as Agentsheets: Icons are located in cells of a grid [48]. The rules are ordered in the same way as ChemTrains, linearly with the possibility of having groups of rules with the same priority. However, instead of executing just one action at a time, all possible actions are executed in parallel as follows. First, all rules are matched against the current situation. Then each rule that matched is execute sequentially if the before-picture is still consistent with the situation. This simple strategy solves a range of problems that require the parallel execution of actions.

## 3.2 Cartoonist Instances

The Cartoonist framework is independent of an actual visual programming system. It describes a family of systems just as the rule-based paradigm describes the systems in the previous section. Cartoonist can even be built on top of existing visual programming systems. An instance of the Cartoonist framework, called a Cartoonist system, consists of at least the following parts.

**Visual Interface** Its main function is to show the simulation and to allow the user to define cartoons, rules, and characters.

**Characters** A character is an object in the simulation that can execute a set of actions, for instance, move around, change its appearance, or make a sound.

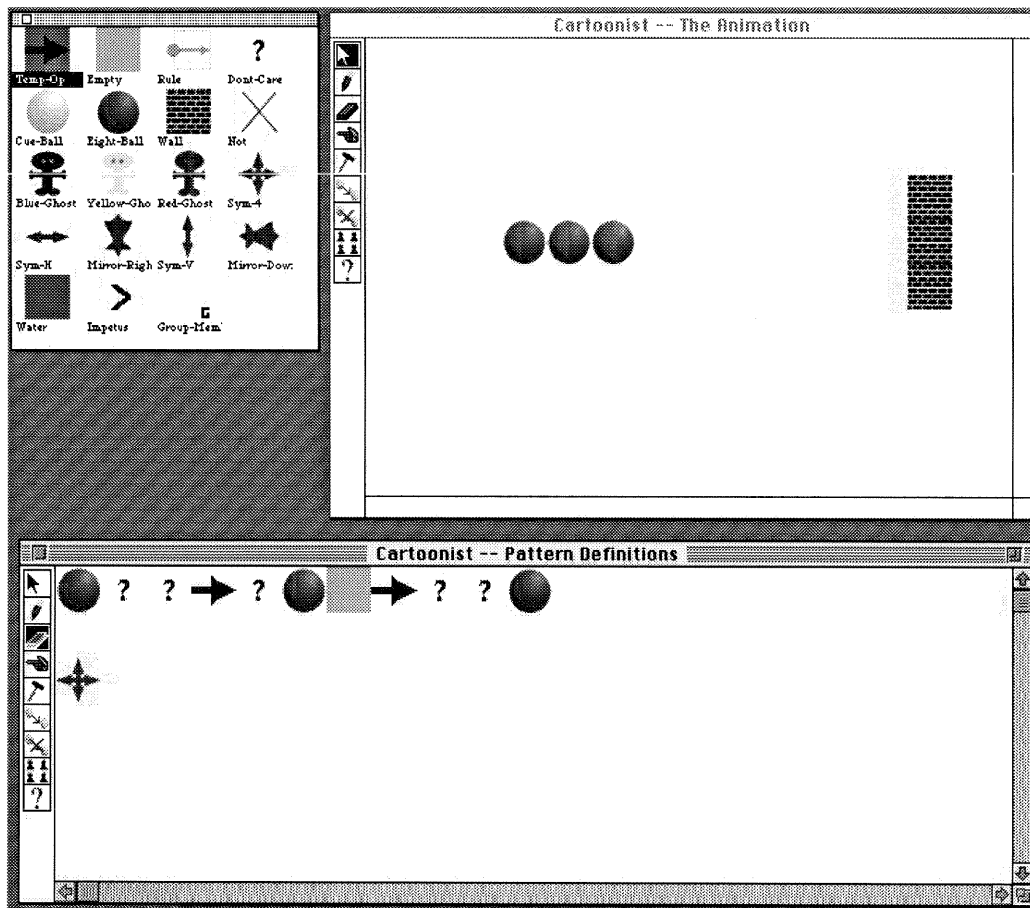
**Rules** Rules add actions to a character created by the user.

**Cartoons** Cartoons describe the characters' behavior and their interaction with each other.

**Rule Language** An OPS5-like language describes conditional rewrite rules or condition-action rules.

**Cartoon Language** Cartoons are described with a constraint language.

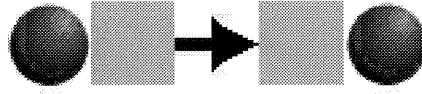
**Cartoonist Engine** The engine coordinates the rules and cartoons and decides what action is to be executed next. The relations do not need to be primitive relations.



**Figure 3.6:** The Agentsheets-Cartoonist programming environment consisting of the character palette, the simulation display, and the programming window to define rules and cartoons.

The visual interface shown in Figure 3.6 displays the simulation in the *simulation display*. The simulation display contains the visual objects, called *characters*. A character is not just an icon, but has actions associated with it. For instance, in a particular model a ball can move in any of the four directions: up, down, left, and right. However, it cannot move to a place occupied by another character.

The term visual *interface* is somewhat misleading, as it implies that it is merely a medium through which the user accesses the “real thing.” With Cartoonist the user works *in* the interface, and the interface is all that exists for the user. The program is created in the same medium as it is run, namely in the visual interface.



**Figure 3.7:** Actual snapshot from ASC. The rewrite rule moves the ball to the right. The gray square means that this square must be empty. The pattern on the left side of the arrow must match the current situation. If this rule is executed, the matched part is replaced by the pattern on the right side of the arrow.



**Figure 3.8:** Same rewrite rule as Figure 3.7 with the grid made visible.

### 3.3 Agentsheets-Cartoonist

The prototype of Cartoonist in Figure 3.6, called Agentsheets-Cartoonist (ASC) is built on top of Agentsheets, a grid-based environment well suited to implement visual languages [63]. The Agentsheets environment provides an editor to create the visual representation of characters, called depictions. In the top left corner of Figure 3.6 is the palette of characters accessible to the user. The top right window shows the running simulation in the simulation display, and in the window at the bottom the rules and cartoons are defined.

Throughout this thesis, I will use examples that have visual representations as used in Agentsheets-Cartoonist. Many of the pictures are actual snapshots of the prototype and sometimes they are created with a drawing tool to clarify certain issues that cannot easily be seen in ASC's interface. In Agentsheets, agents are located in a two-dimensional invisible grid. Thus, every agent (or character in ASC terms) has four "normal" neighbors (east, west, north, and south) and four diagonal neighbors.

The pictures in the rules and cartoons are also grid-based. For instance, the rewrite rule in Figure 3.7 directs the ball to move to the right onto the empty square, if the condition of the rule matches. Figure 3.8 shows the same rule, but with a visible grid. The picture on the left of the arrow is the before-picture; the after-picture is on the right of the arrow. In a rule and in a cartoon, an empty, white square means that it does not matter whether the square is occupied by another character. Since the grid is normally not shown, a question mark is used to make explicit that a certain square is a "don't-care" square. A gray square as in Figures 3.7 and 3.8 means that the square must be empty. Of course, in the simulation display an empty square is white.

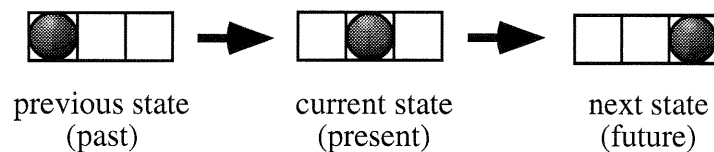
Every character has an identity, that is, each gray ball in the simulation is unique. In Figure 3.7, the ball in the before-picture and the ball drawn by the after-picture one square to the right have the same identity. This is not immediately visible in the static picture



**Figure 3.9:** This cartoon makes the ball move to the right despite what the ball has done before. The question marks mean that there is no constraint on this square.



**Figure 3.10:** This cartoon specifies how a ball keeps moving to the right by describing three states that follow each other. This figure is a snapshot from ASC. Figure 3.11 shows the same cartoon.



**Figure 3.11:** Same cartoon as Figure 3.10, but the association of the constraints with the past, present, and future states are more explicit.

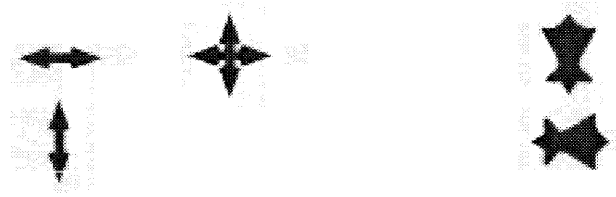
of the rule. The relation between the two balls is realized in the construction of the rule, specifically by copying the ball from the before- to the after-picture.

As described earlier, a cartoon has a form similar to that of a rule; however, its meaning is quite different. A cartoon consists of at least two state descriptions (or constraints). As noted earlier, each constraint is matched against a state, such that the last constraint describes the next state in the immediate future, the second-to-last constraint describes the current state, and the constraints preceding the current state constraint describe states in the past.

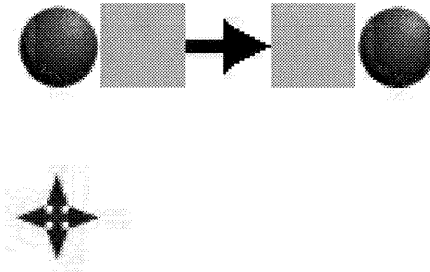
Assume we want to describe how the ball keeps moving to the right. This is described with a cartoon, because it is a behavior and not an action. The solution is the cartoon in Figure 3.9, which is similar to the rewrite rule in Figure 3.7. However, the problem with this cartoon is that it makes every ball move to the right, independent of a ball's previous movements.

The improved solution is shown in Figure 3.10, which describes a ball moving to the right if it has been moving to the right in the immediate past. The picture  $? \bullet ?$  in the middle of the cartoon  $\bullet ? \rightarrow ? \bullet ? \rightarrow ? ? \bullet$  describes the current state, the picture on its left the previous state, and the picture on its right describes the next state. This is made more explicit in Figure 3.11.





**Figure 3.12:** The number of rules and cartoons that have to be drawn by the user can be dramatically reduced by taking advantage of symmetries in the grid. See text for the explanation of each icon.

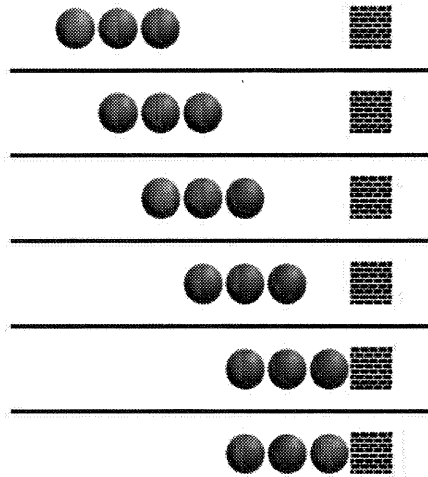


**Figure 3.13:** The same rule as in Figure 3.7 now rotates in all four directions, that is, this rule actually represents four rules.

So far, balls can only move to the right. It would be easy to add additional rules and cartoons, one for each direction. But that would be cumbersome for most users, although participants of the Child’s Play workshop [37] have suggested that children do not seem to mind much, at least initially. The user can take advantage of the icons shown in Figure 3.12.

The icon  $\leftrightarrow$  adds the current rule or cartoon and the one rotated by  $180^\circ$ ; that is, the move-right rule plus this icon would result in a move-right rule and in a move-left rule. The icon  $\ddagger$  rotates  $90^\circ$  and  $270^\circ$  but does not add the original rule or cartoon, resulting in a move-up rule and a move-down rule.  $\star$  combines the two previous icons, resulting in moves in all four directions. The rule in Figure 3.13 generates move actions in all four directions. Finally,  $\blacktriangleleft$  mirrors the rule or icon horizontally and  $\blacktriangleright$  does the same vertically. Additional elements will be introduced in the following examples when they are needed.

The Agentsheets-Cartoonist prototype provides a grid-based visual interface that influences the possible relations that can be used to describe a picture. Only local relations can be expressed, for instance, whether one object is on the south-, north-, west-, or east-side of another object. Often, a cartoon for a behavior in some direction can be easily generalized into other directions using the operators introduced above.



**Figure 3.14:** Rolling balls problem. The major problem here is to make all the balls move at once. No existing rule-based visual system is able to solve this problem correctly.

## 3.4 Examples of Simulations

The following examples of simulations illustrate the unique features of Cartoonist. All simulations except the PacMan example are implemented in the Agentsheets-Cartoonist prototype.

### 3.4.1 Rolling Balls

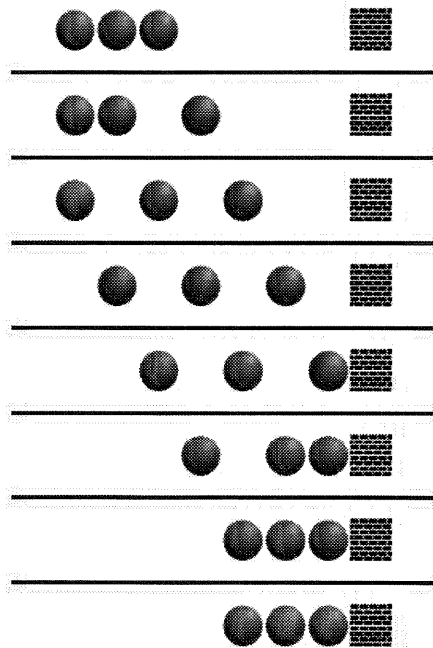
This task is similar to the balls moving in one direction as shown in the previous sections. However, the balls are in a row next to each other. In Figure 3.14 the balls roll in the same direction and stay together. The wall stops the ball chain. The solution must be general enough that it works in any of the four directions and with any number of balls in the chain.

This problem is difficult for existing rule-based systems because it requires the parallel execution of a set actions that is larger than the set of possible actions in the situation.

#### 3.4.1.1 Attempts with Rules

Users and creators of purely rule-based visual systems have attempted to solve the rolling ball problem. Most of the attempts used the move-right rule in Figure 3.7 and a system such as ScienceShows, which basically applies all possible actions in parallel. ScienceShows' rule-application algorithm, described in Section 4.2.2, is slightly more complicated, but this does not influence the solution to the rolling balls problem.

Assuming the balls move to the right, a ball moves if the square is on its right is empty. The resulting behavior is shown in Figure 3.15 and makes the difficulty of the problem



**Figure 3.15:** ScienceShows’ solution to the rolling balls. The balls have to spread out before they all can move.

apparent: How can a ball in the chain move if it is not clear that the ball in front of it moves, too? Not all actions that have to be executed in parallel can fire in this situation.

This suggests that the problem might be solved by relaxing the condition of having an empty square in front of a moving ball. If the rule says “go to the right no matter what’s on the next square” then the ball chain cannot be stopped by anything and it crashes through thick walls, as in Figure 3.16.

This leads to a more advanced version of the no-matter-what rule, which says something like, “Move to the right if the square is empty or is occupied by another ball.” This does not work either, as the simulation in Figure 3.17 shows. It is just a trick and not an acceptable solution. Balls cannot move to a place that is already occupied by another physical object, whether it is a ball or a wall.

An alternative “solution” was suggested by Brigham Bell using his ChemTrains system. Write a rule that moves the last ball in front of the first ball (see Figure 3.18). This cannot be accepted as a solution of the original problem, because the balls do not move as a chain. Furthermore, it works only for ball chains with exactly three balls.

All the solutions so far suffer from a further problem. The rules do not consider direction. Adding it would not be difficult, but it would be cumbersome, requiring the user to add directional features to the ball, for instance an arrow or a pointed nose.

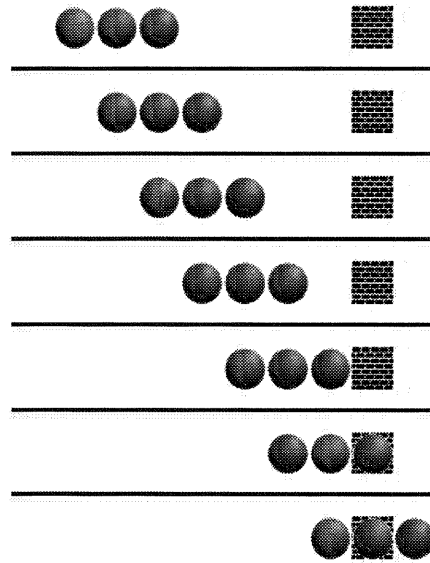


Figure 3.16: If balls can move to any square then nothing can stop them, not even a wall.

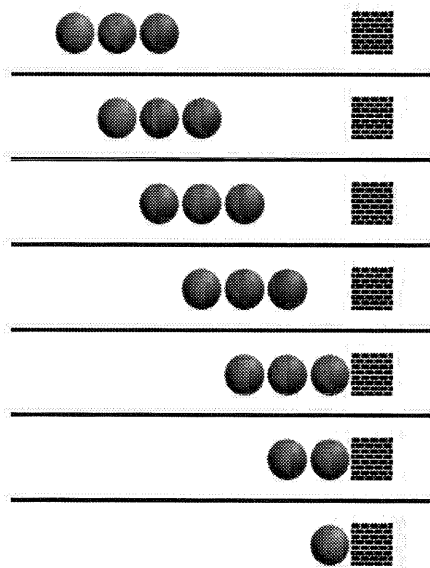


Figure 3.17: Yet another failed attempt. All the balls stack up next to the wall.

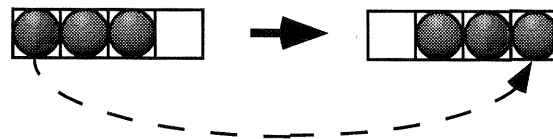
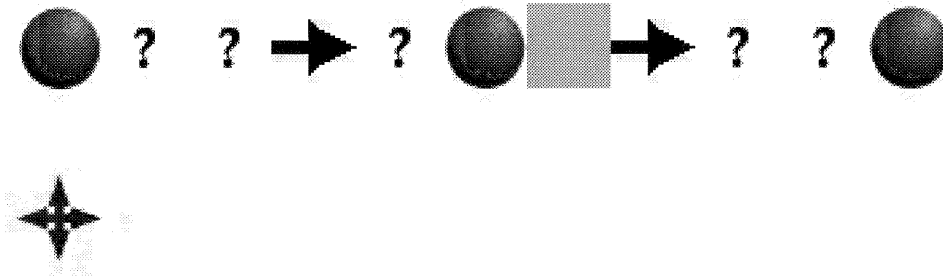
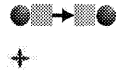


Figure 3.18: This rule suggested by Brigham Bell makes the balls *appear* to move to the right together. Although an interesting way to make it look right, the balls do not really move like a chain, and the solution is not general enough. The dotted arrow shows how the front ball moves to the back.




**Figure 3.19:** The first ball of the chain is moving.

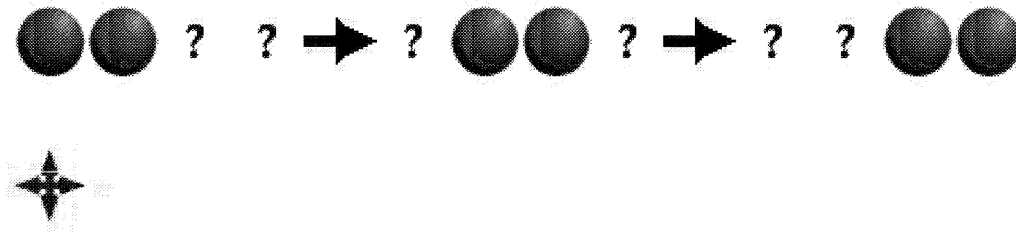
### 3.4.1.2 Solution with Cartoons

The solution with cartoons is simple and relies on several characteristics of the Cartoonist framework. In a Cartoonist system characters are not simply icons. Characters have actions they can execute if a cartoon describes an appropriate state sequence. (For a detailed description see Chapter 2.) A ball can move in any of the four directions provided the square it moves to is empty. These actions are described by the rule  in Figure 3.13. Therefore, when drawing the cartoons, the user need not worry whether the ball might do something impossible, because the definition of the actions prevents the ball from doing anything impossible. All the user has to do is describe the current state, the next state, and, if needed, one or more past states.

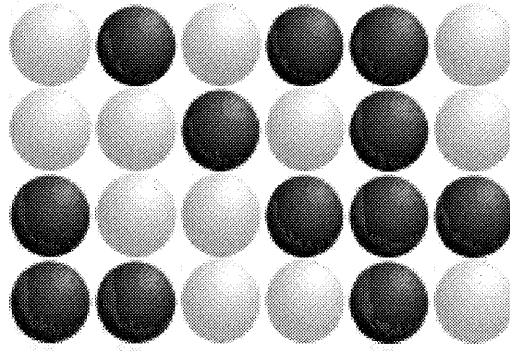
The second useful characteristic of cartoons is that they can be used to describe temporal concepts. Using the same idea as in the cartoon in Figure 3.10, the ball chain can be given a momentum that keeps it moving in the same direction as it moved in the past as long as it does not bump into something. Furthermore, cartoons are used such that the actions are executed in parallel whenever possible. However, the user does not need to think about parallel execution or even different kinds of parallelism.

The Cartoonist solution consists of two parts: (1) describing what the first ball in the chain does and (2) describing what the other balls do. The first ball may move if the square in front of it is empty, where the direction is determined by the previous position of the ball in its immediate past. The cartoon describing this is shown in Figure 3.19. The picture  describes the current state.

Any other ball in the chain has no empty square in front of it. However, it can move if its neighbor in the square in front of it moves, too. In other words, pairs of balls can move, too. The necessary cartoon is shown in Figure 3.20 and works because a row of  $n$  balls can be viewed as a row of  $n - 1$  overlapping pairs of balls. The second and the last picture in Figure 3.20 describe how a pair changes its location from the current state to the next state.



**Figure 3.20:** Any pair of balls in the chain can move. These are just snapshots and do not define a new action.



**Figure 3.21:** An initial situation of the Game of Life as well as of the simplified version 'Life'.

### 3.4.1.3 Summary

Rule-based systems that execute actions in parallel cannot solve the problem of the rolling balls adequately because they require that all potential actions must be known in the initial state. In the initial state, however, all but the first ball are stuck, because a ball can move only if the ball in front of it moves.

The declarative nature of the cartoons makes it possible to describe how the different states look. The Cartoonist engine makes sure these states can be reached if the actions are executed in parallel. This works well in the current case, because a ball can move forward if the ball in front of it moves too.

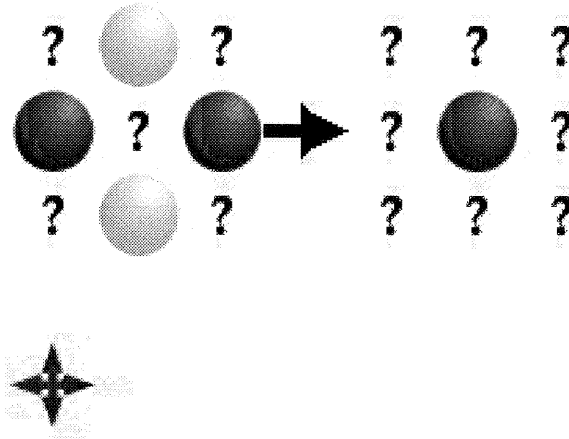
## 3.4.2 Game of Life

The *Game of Life* consists of a living and dead cells arranged in a grid as in Figure 3.21, each cell having eight neighbors. A living cell will be alive in the next generation if it has two or three living neighbors. A dead cell becomes alive if it has exactly three neighbors. In all other cases, the cell will be dead in the next generation.

It is important that all actions are executed in parallel. If the actions are executed sequentially, they may change the neighborhood of a neighbor cell and thus, the neighbor



**Figure 3.22:** The rules specifying a cell's actions: A cell can become alive (left rule) and it can die (right rule).



**Figure 3.23:** A cell is alive in the next generation if it has currently exactly two living neighbors independently of whether the cell is currently living or dead.

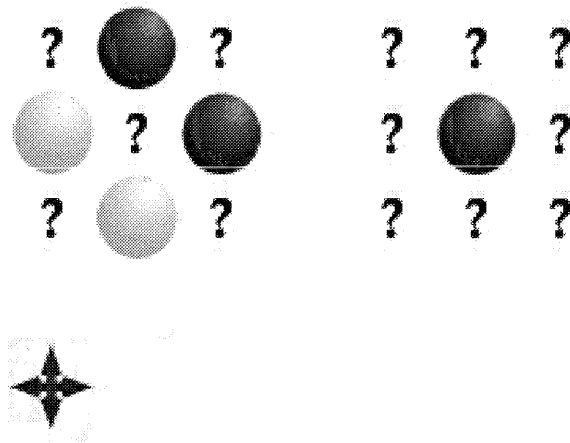
cell may possibly do the wrong thing. A completely parallel execution strategy is therefore necessary.

A rule-system with a completely parallel execution strategy solves therefore this problem relatively easily with the rules

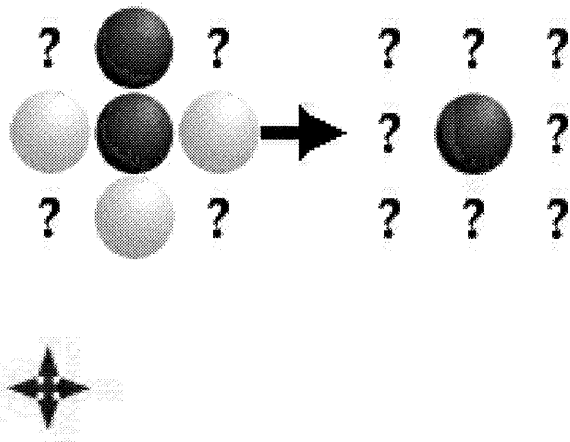
1. { the cell is alive and has 2 or 3 neighbors }  $\rightarrow$  { the cell is alive }
2. { the cell is dead and has 3 neighbors }  $\rightarrow$  { the cell is alive }
3. { the cell exists }  $\rightarrow$  { the cell is dead }

The Cartoonist solution is similar. Since displaying all the necessary cartoon would take up too much space, a slightly simplified version *Game of Life'* is described. In *Life'*, a cell has only the four neighbors sharing a side of the cell's square. A cell stays alive if it has one or two living neighbors and it becomes alive if it has exactly two neighbors. Otherwise, the cell dies.

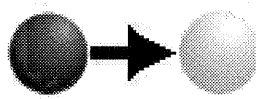
The actions a cell can execute is to become alive and to die (see Figure 3.22). The cartoons in Figures 3.23–3.26 the cells' behavior according to the game's rules. The last cartoon in Figure 3.26 must have a lower priority, because it may only apply if the other cartoons do not. This means, that the last cartoon plays the same role as a catch-all clause in a Prolog program.



**Figure 3.24:** This cartoon describes the same behavior as in Figure 3.23 except that the two living cells are not placed across each other.



**Figure 3.25:** A cell stays alive if it has exactly one living neighbor.



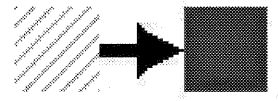
**Figure 3.26:** The cell dies. This cartoon must have a lower priority than the other cartoons.

Assuming that the rule-based system uses the completely parallel execution strategy, the solutions of the rule-based system and the Cartoonist system are basically the same.

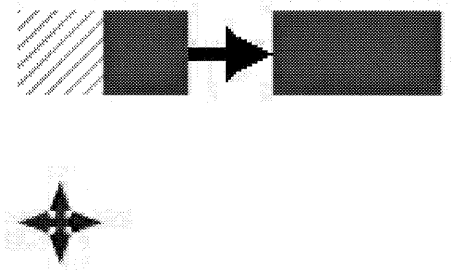
### 3.4.3 Melting Ice

The melting ice problem shows consists of one or more ice blocks. The ice melts along the surface and therefore, an ice block must melt slower than the same amount of crushed ice, because the latter has the greater surface. A simulation of melting ice is showing in





**Figure 3.27:** Rule implementing the action that ice can transform into water. The dark gray square stands for water, not an empty square.



**Figure 3.28:** Ice melts when it is next to water. There is no need (nor possibility) to say something about the kind of parallelism used in this case as, for instance, in Agentsheets.

Figure 3.29. This problem was suggested by ScienceShows' parallel execution algorithm, which solves this problem easily.

A rule-based system using the same execution strategy as in the Life example can easily get into trouble. Assume that the rule says that an ice piece melts if it is next to a piece of water. An ice piece at the corner of the ice block has two neighbors that are water. Thus, the ice melts twice.

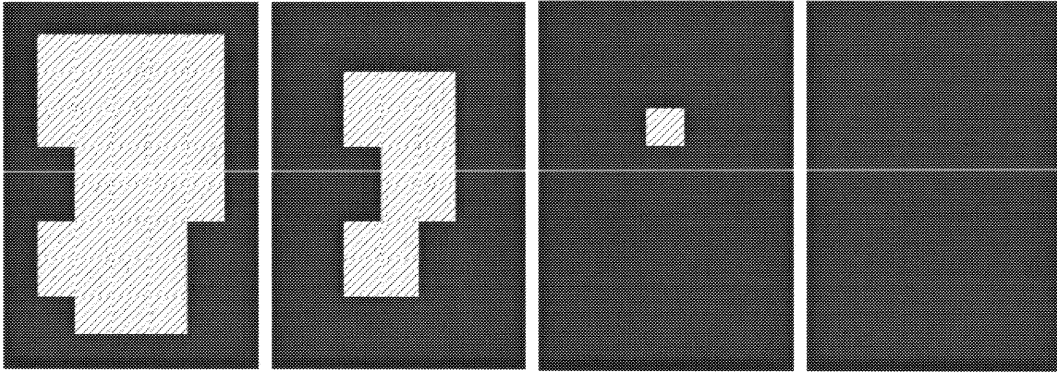
The parallel execution strategy that solves this problem always test the condition of the rule again, before the rule is actually executed. In this case, once the first action has melted the ice piece, the condition of the second rule does not apply anymore. This execution strategy is more useful since not many problems require a completely parallel execution of the actions.

The Cartoonist solution requires one rule and one cartoon. The rule in Figure 3.27 implements that ice can be transformed into water. The cartoon in Figure 3.28 describes that ice melts if it is next to water.

The actual program, the cartoon in Figure 3.28, leads to the simulation in Figure 3.29. The cartoon looks the same as the rule drawn in ScienceShows for the same problem [48].

### 3.4.3.1 Summary

Assuming the ice character already exists in the palette, the Cartoonist solution is as simple as the solution in ScienceShows or the rule version of Agentsheets using the same parallel execution algorithm as ScienceShows.



**Figure 3.29:** Ice melts along the surface as, the snapshots show from left to right.

The rule-based system that worked well for the Game of Life does not melt ice correctly. Furthermore, the execution strategy that works for the ice melting problem would lead to an incorrect behavior in Game of Life example. These problems do not arise in Cartoonist, which is a great advantage, because the user does not need to decide which kind of parallelism to choose.

### 3.4.4 Collisions

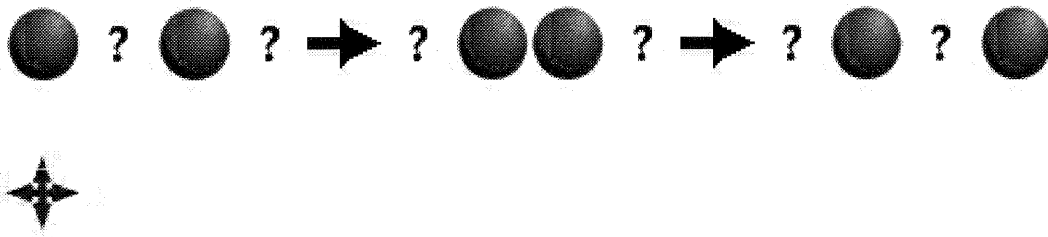
In this example, balls bump into each other and bounce back from walls. A ball that bumps into a resting ball is completely stopped and the other ball starts moving to the same direction.

This interaction between the two balls is problematic for rules because they can only describe the transition between two states. However, the interaction consists of at least two actions in sequence and thus, three states: The first ball moves towards the resting ball and then, the previously resting ball moves.

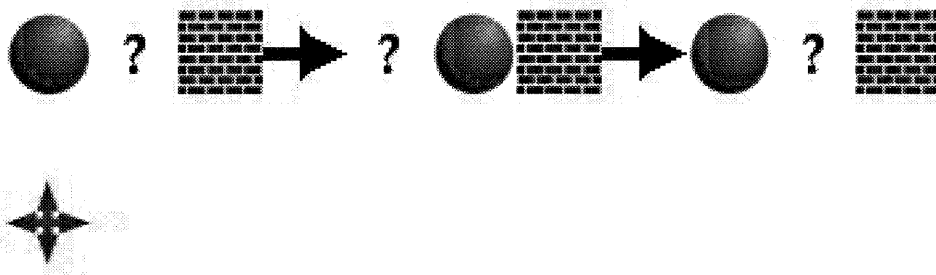
cartoons can describe state sequences of any length. The cartoon in Figure 3.30 describes this interaction. A ball that bumps into a wall reverses its direction and keeps moving. The situation before, during, and after the bump describes the whole interaction and is shown in Figure 3.31.

The two cartoons in Figures 3.30 and 3.31 describe the whole process. A few snapshots of the balls bouncing into each other are shown in Figure 3.32. No additional graphical representation of the direction is necessary.

The first cartoon in Figure 3.30 describes the collision of the two balls. The first and the second picture describe how the ball coming from the left bumps into the resting ball and the second and third picture describe how the right ball is accelerated while the left



**Figure 3.30:** When a ball collides with another ball of the same kind, the former ball can be stopped and the latter starts moving.



**Figure 3.31:** A ball bumps into a wall can be simply described with three snapshots.

ball stays at the place of collision. The second cartoon in Figure 3.31 is similar to that in Figure 3.30, except that the wall is not accelerated but the ball's direction is reversed.

#### 3.4.4.1 Summary

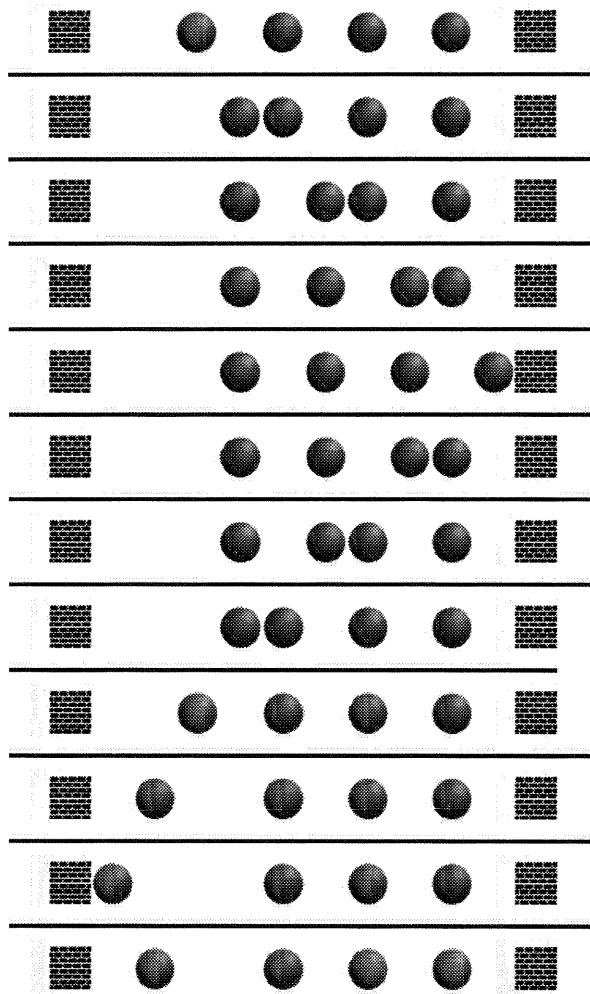
It is not always easy to figure out the conditional actions underlying an interaction among several objects as the history of particle physics tells us. Cartoons allow the user to describe an interaction as it is observed without requiring the user to find the necessary conditional actions causing the correct behavior. This can be a great advantage if the user is interested in the resulting behavior only. If the user looks for a “deeper” model based on the conditional actions, then he or she has to resort to rules instead which is also possible in Cartoonist.

Quite often, the description of interactions and temporal concepts can be combined in a cartoon. For instance, the two cartoons used in this example describe to what behavior the impetus can lead in different situations.

### 3.4.5 Negative Programming: Avoiding Each Other

The balls in Figure 3.33 must avoid each other, yet they keep their direction and bump off walls correctly as long as they do not end up next to another ball.

This problem requires many non-trivial rules in a rule-based system. In Cartoonist, it is rather simple.



**Figure 3.32:** The cartoons in Figure 3.30 and Figure 3.31 completely describe the process shown as a sequence of snapshots.

Sometimes it is easier to describe behavior by stating what should *not* happen. Since cartoons are declarative, negation can be used in the description of the future state. In this example I will also show how the behavior of the characters can easily be refined.

Assume that the balls in Figure 3.33 are moving around randomly and that they should avoid each other. Moving around randomly can be expressed succinctly with the cartoon in Figure 3.34. The cartoon says that in the next state the ball will not be at the location it is now. This is expressed by the meta-character “x” crossing out the ball in the second picture. Since a character, for instance a ball, in a picture means “a ball is at this location” the crossed out ball means “the ball is not at this location.” Describing that the balls should avoid each other can be done by saying that a moving ball should not end up next to another ball. The cartoon in Figure 3.35 describes this.

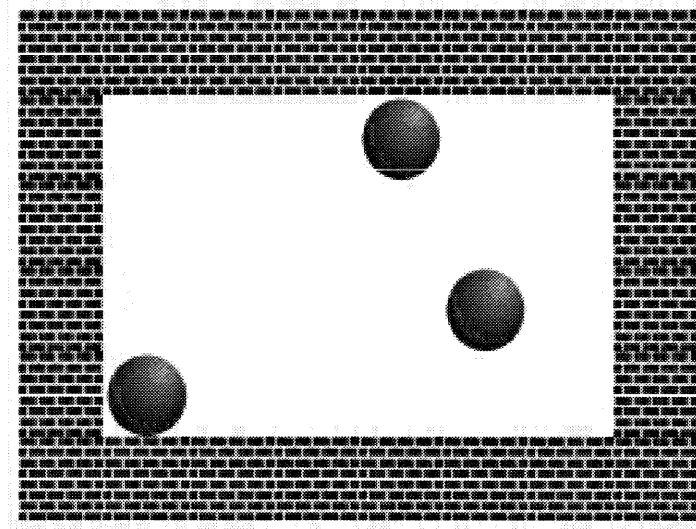


Figure 3.33: Snapshot of balls avoiding each other.

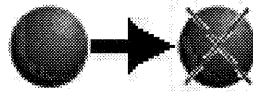


Figure 3.34: Cartoon describing moving around as “don’t stay at the same place.” The crossed-out ball means that the ball must not be there. What is invisible, and a problem of visual notation, is that the two balls are the same at different times.

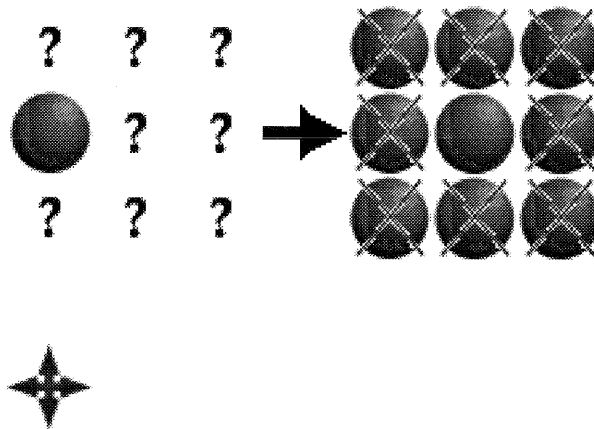
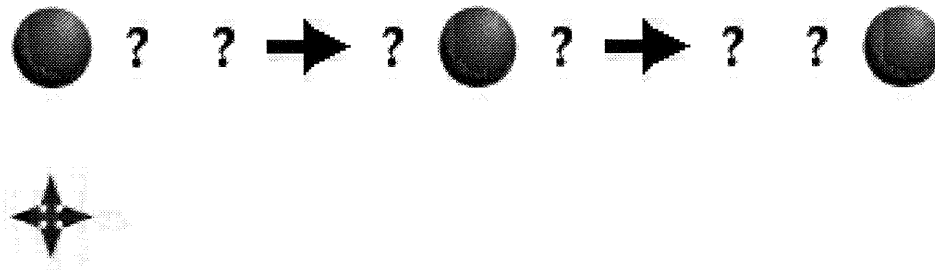


Figure 3.35: Cartoon showing that the moving ball does not end up next to another, possibly moving, ball. The cartoon has to be rotated and mirrored to capture all possible cases.

Watching the simulation, the user is not too happy with the result because the balls behave as if they were drunk. They indeed avoid each other, but it would be good if they had some direction in their movement. The solution is to add the cartoon in Figure 3.36, which keeps the ball moving in the same direction. The preference is set such that the



**Figure 3.36:** A ball, once moving into a certain direction, keeps moving into the same direction.

cartoon describing that the balls avoid each other is preferred over the new cartoon. The resulting behavior is already much more appealing.

However, the balls do not bounce back from the walls correctly. The solution is simple: add the cartoon from Figure 3.31. Again, the avoid-balls cartoon is preferred over the new one. Now, the balls still avoid each other, but they bump back from walls and keep going straight if possible.

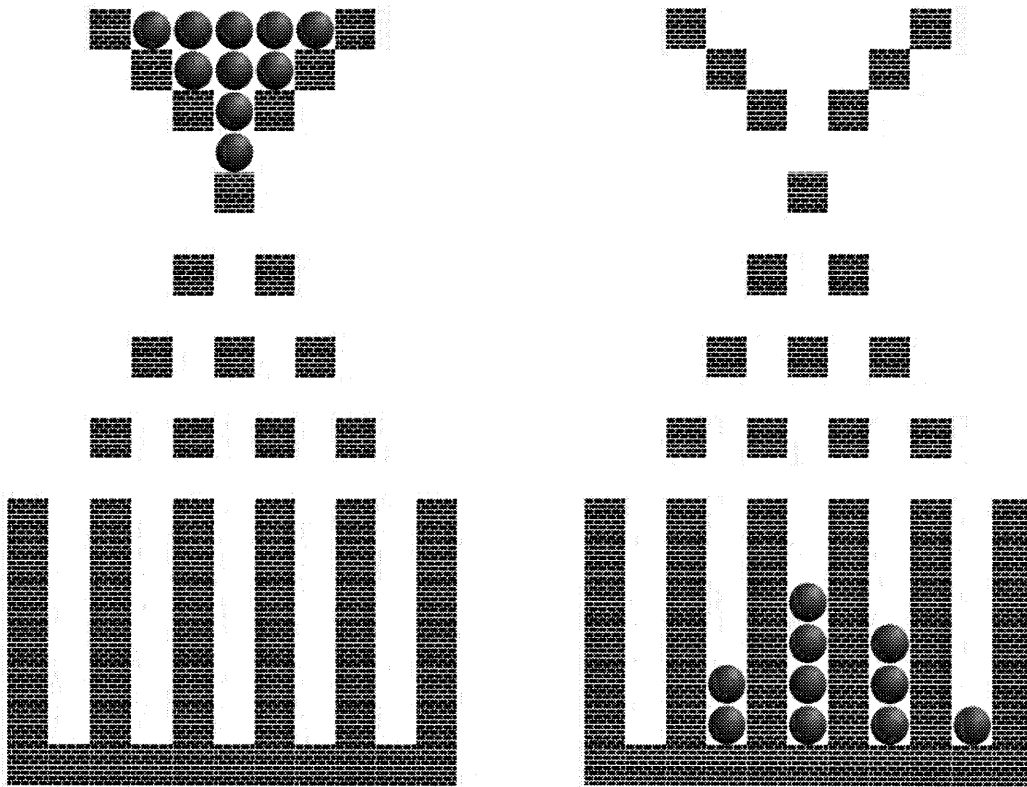
#### 3.4.5.1 Summary

Programming using negation in the after-picture can simplify describing certain behavior. Rules can use negation in the condition; however, describing the behavior in this example with rules would be tedious, because the whole situation must be tested in the condition to figure out the right action. A cartoon can be used to specify what the action has to achieve, whatever action it is.

Generally, a cartoon using negation in the after-picture can be satisfied with many actions, which is useful if one wants to further refine the behavior of the characters. This allows the user to specify the behavior of the characters given certain additional constraints. The subsumption-like architecture is useful for these kinds of specifications, as the example shows. The balls move around as one would expect *under the constraint that* they never do touch each other. What is so nice is that the constraint can be stated completely separate from describing the rest of the behavior.

### 3.4.6 Pachinko Task

At the first Child's Play workshop [37], the Pachinko task in Figure 3.37 was suggested as a test example for end-user programming systems. The balls fall down and when they hit a pin, represented as a wall piece, they drop down either on the left or right of the pin. The



**Figure 3.37:** On the left is the initial situation and on the right is the final solution after all the balls have fallen down.

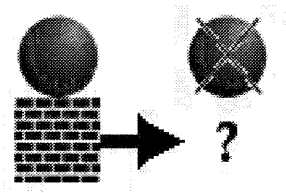


**Figure 3.38:** Gravity is described with two cartoons. The one on the left says “don’t move up”; the one on the right describes that a ball falls down if it is not supported. (The gray square describes an empty square.)

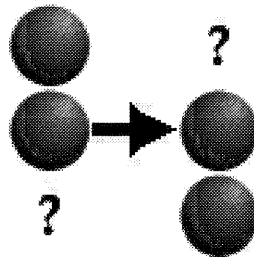
resulting distribution at the bottom of the figure should approximate a normal distribution if enough balls are dropped.

All the rule-based visual systems did well, except that they required the balls to spread out as demonstrated in the rolling balls example at the beginning of this section.

In the Pachinko task, balls are dropped onto a triangle of pins, in Figure 3.37 depicted as little brick-wall pieces, resulting in an approximation of a bell curve at the bottom. AS-Cartoonist’s solution consists of four simple cartoons. The cartoon describing that a ball never may move up and the cartoon saying that an unsupported ball drops (see Figure 3.38)



**Figure 3.39:** A ball on a wall rolls around “searching” for a gap. This is described by saying the ball does not stay at the same place.



**Figure 3.40:** The behavior that a ball chain may drop as a whole.

take care of gravity. The cartoon in Figure 3.39 makes sure that balls do not get stuck on brick walls; that is, a ball keeps moving until it finds a gap to fall down.

These cartoons lead to the same behavior as displayed by the other systems at the Child’s Play workshop. However, these solutions miss the fact that ball chains can fall together, so the cartoon in Figure 3.40 is added, which leads to a parallel and correct behavior.

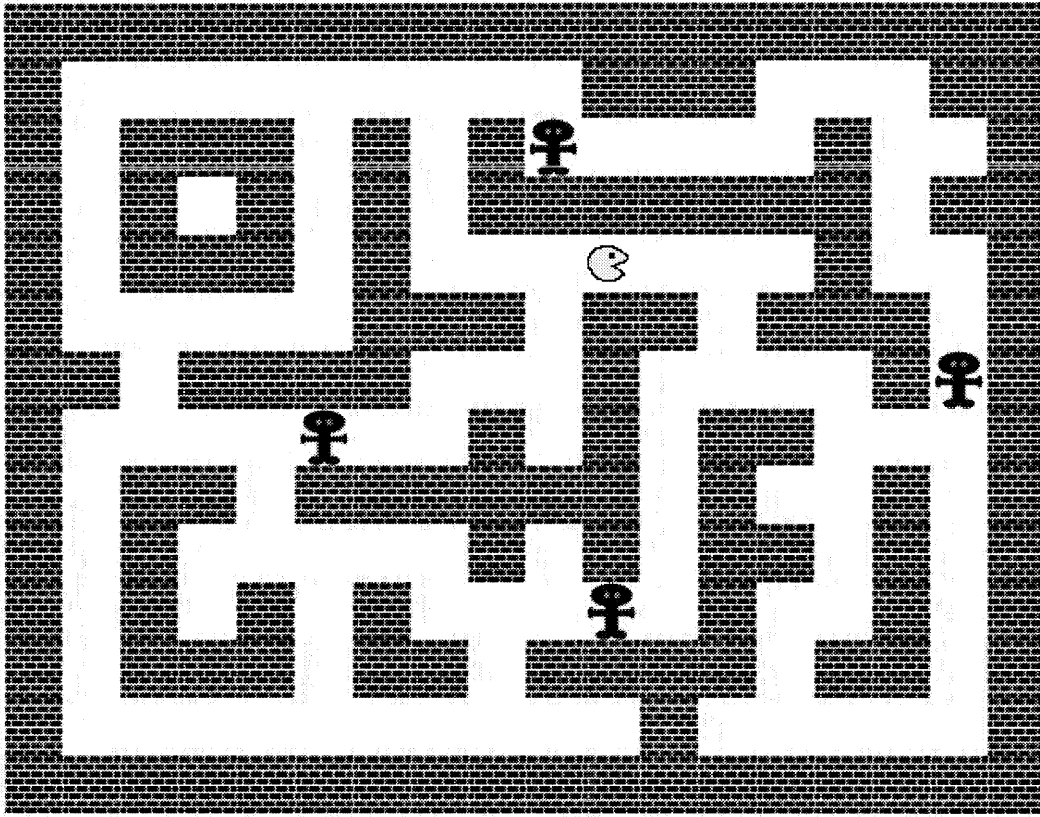
### 3.4.6.1 Summary

The Pachinko example integrates several of Cartoonist’s unique characteristics. The balls can fall down together whereas the other systems either can execute only one action at a time or require the balls to be spaced out to fall down in parallel. Negative programming was again be used in an effective way to prevent a ball from getting stuck on a wall piece and from moving upward, thus defying the law of gravity. Finally, the possibility to describe complex behavior by combining simple descriptions has been useful. For instance, the cartoon in Figure 3.40 was added later without having to change any of the other cartoons.

### 3.4.7 PacMan

The PacMan example shows the strength of the declarative nature of cartoons and makes clear another advantage of separating what a character can do implemented with rules from the description of its behavior described with cartoons.





**Figure 3.41:** The ghosts chase the PacMan in the maze.

In the game of PacMan, several ghosts chase the PacMan in a maze (see Figure 3.41). The task is to make the ghosts track the PacMan, assuming the rules and cartoons have access to the ghosts' and the PacMan's coordinates. First, a maze is assumed where the characters can move in only four directions, east, west, north, and south. Then the solution has to be extended to include the diagonal directions as well. Because the grid-based approach of ASC is not well suited for describing global relations such as the distance between two remote characters, a purely textual representation of the rules and cartoons is used to be able to compare the solutions.

A character and its location is described by the relation  $at(\langle type \rangle, \langle id \rangle, \langle x \rangle, \langle y \rangle)$ , where  $\langle type \rangle$  is the character's type, which is either 'ghost' or 'pacman';  $\langle id \rangle$  is the unique identifier of the character; and  $\langle x \rangle$  and  $\langle y \rangle$  are the characters' coordinates in the grid. For each empty square there is a predicate 'empty.' Variables are written in *italics*.

### 3.4.7.1 Solution with Rules

First, a solution with rules is presented to which the cartoon solution can be compared. Having access to the characters' coordinates leads to an obvious solution. The rules  $R_1, \dots, R_4$

move the ghost closer to the PacMan, depending on the location of the PacMan with respect to the ghost. It would be easy to generate rules  $R_2, \dots, R_4$  from rule  $R_1$  and information about the symmetries of the rule, reducing the need for drawing all four rules.

Rule  $R_1$  says if the ghost  $g$  is on the left of the PacMan  $p$  expressed with  $x_g < x_p$  and the square on the right of the ghost is empty, then the ghost moves to the empty square, the pacman stays at its old place, and the square at  $x_g, y_g$  is now empty. The other three rules are similar:  $R_2$  moves the ghost to the left,  $R_3$  moves it up, and  $R_4$  implements the down action.

$$R_1 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ x_g < x_p, \text{empty}(x_g + 1, y_g) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x_g + 1, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

$$R_2 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ x_g > x_p, \text{empty}(x_g - 1, y_g) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x_g - 1, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

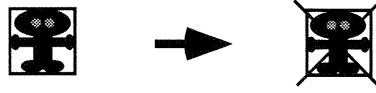
$$R_3 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ y_g < y_p, \text{empty}(x_g, y_g + 1) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g + 1), \\ \text{at(pacman, } p, x_p, y_p), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

$$R_4 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{at(pacman, } p, x_p, y_p), \\ y_g > y_p, \text{empty}(x_g, y_g - 1) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g - 1), \\ \text{at(pacman, } p, x_p, y_p), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

With these rules, a ghost can track the PacMan; however, if it cannot get closer to the PacMan because of a wall, the ghost cannot move and just sits there. To fix this problem, four more rules have to be added. Again, just one is shown for going to the right without getting closer to the PacMan. It is important that the rules  $R_1, \dots, R_4$  are preferred over the rules  $R_5, \dots, R_8$ ; that is, tracking is preferred over random walking. Furthermore, neither of the two rule groups should be preferred over the other because otherwise the ghosts' behavior becomes deterministic and does not look natural to the user.

Rule  $R_5$  says that if the square on the right of the ghost is empty, then the ghost moves there. The rules  $R_6, \dots, R_8$  implement the same action for the other three directions. To represent these rules visually is easy, which is unfortunately not true for rules  $R_1, \dots, R_4$ , because they need to access non-local information.

$$R_5 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{empty}(x_g + 1, y_g) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x_g + 1, y_g), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$



**Figure 3.42:** Visual representation of cartoon  $\alpha_1$ . The crossed-out ghost means the ghost is not at the same place as before, that is, it has moved.

These eight rules solve the first part of the problem. Now the solution has to be extended to include movement along the diagonals. Unfortunately, this requires another eight rules  $R_9, \dots, R_{16}$  similar to the first eight ones resulting in a total of sixteen rules. Again, only one of the two sets of rules is shown.

$$R_9 : \left\{ \begin{array}{l} \text{at}(\text{ghost}, g, x_g, y_g), \\ \text{at}(\text{pacman}, p, x_p, y_p), \\ x_g < x_p, y_g < y_p, \\ \text{empty}(x_g + 1, y_g + 1) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at}(\text{ghost}, g, x_g + 1, y_g + 1), \\ \text{at}(\text{pacman}, p, x_p, y_p), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

$$R_{13} : \left\{ \begin{array}{l} \text{at}(\text{ghost}, g, x_g, y_g), \\ \text{empty}(x_g + 1, y_g + 1) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at}(\text{ghost}, g, x_g + 1, y_g + 1), \\ \text{empty}(x_g, y_g) \end{array} \right\}$$

The rules should be ordered as follows to get a correct and natural behavior.

$$\{R_1, \dots, R_4, R_9, \dots, R_{12}\}, \{R_5, \dots, R_8, R_{13}, \dots, R_{16}\}$$

The rules within a set are not preferred over each other, but the rules in the first set are preferred over the rules in the second set.

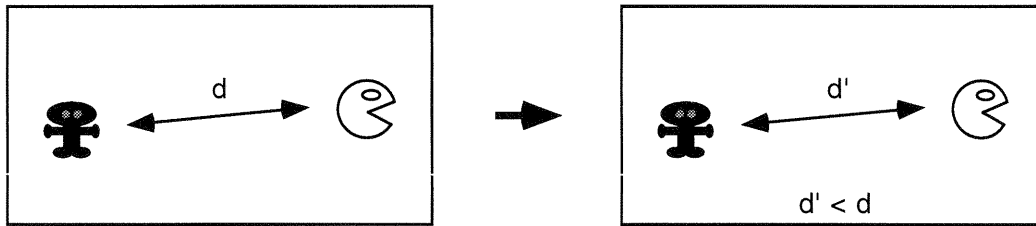
The rule-based solution is not only problematic, because many rules are required, but also because the rules combine information about the ghost's strategy and what action it has to execute to implement the strategy. Therefore, changes in what actions a ghost can execute requires changes in the rules that implement the tracking strategy.

### 3.4.7.2 Solution with Cartoons

The first cartoon  $\alpha_1$  (shown visually in Figure 3.42) directs the ghosts to move around. As long as there is not a more specific description, this movement will be random. Because of the declarative nature of cartoons, all that has to be said in  $\alpha_1$  is that “the ghost will be at a different place.” Instead of four rules, one cartoon is needed. ACS does not provide access to global information. Therefore, the cartoons are drawn in a possible extension to the visual language used in ACS.

$$\alpha_1 : \{\text{at}(\text{ghost}, g, x_g, y_g)\} \rightarrow \{\neg \text{at}(\text{ghost}, g, x_g, y_g)\}$$

The ghosts do not just walk around randomly, they also track the PacMan when possible. Cartoon  $\alpha_2$  (shown visually in Figure 3.43) describes the tracking behavior by saying that



**Figure 3.43:** The cartoon  $\alpha_2$  drawn in an extension of ACS visual language. The arrow between the two characters refers to their distance, and the expression  $d' < d$  in the second picture is a constraint on the distances.

“the ghost will be closer to the PacMan in the next state.” Again, one cartoon has “replaced” four rules. Furthermore, the user does not need to worry about preferences. An action that gets a ghost closer to the PacMan receives two votes, whereas the other actions get one vote (or zero if the ghost would bump into a wall). The voting scheme implementing the subsumption-like architecture of cartoons is described in Section 2.6.

$$\alpha_2 : \left\{ \begin{array}{l} \text{at(ghost, } g, x_g, y_g), \\ \text{at(pacman, } p, x_p, y_p) \end{array} \right\} \rightarrow \left\{ \begin{array}{l} \text{at(ghost, } g, x'_g, y'_g), \\ \text{at(pacman, } p, x'_p, y'_p), \\ (x'_p - x'_g)^2 + (y'_p - y'_g)^2 < \\ (x_p - x_g)^2 + (y_p - y_g)^2 \end{array} \right\}$$

Now this solution has to be extended to the case where the characters can also move along the diagonals. The ghosts are replaced with ghosts that can move in all eight directions. This is absolutely all that is needed, because  $\alpha_1$  makes sure the ghost moves in *some* direction and  $\alpha_2$  describes that the ghost ends up closer. Both cartoons are independent of the actual actions the ghosts and the PacMan can execute.

### 3.4.7.3 Summary

The separation of what a character *can do* from what it *should do* in Cartoonist helps the PacMan example. If restricted to rules, the tracking behavior and moving-action have to be combined in one rule. This is not necessary if the cartoon in Figure 3.43 is used which simply expresses that the distance gets smaller without referring to the ghosts action.

The distinction between rules and cartoons works so well, because the cartoons are declarative and the procedural rules implement the cartoons’ specifications. This is apparent in both cartoons  $\alpha_1$  and  $\alpha_2$ .

## 3.4.8 Examples: Conclusions

This collection of examples shows that the Agentsheets-Cartoonist prototype has greater functionality than some prominent other visual rule-based systems. Certain kinds of parallel

behavior cannot be programmed with these other systems. The descriptions in ASC with the cartoons do not seem to be more complicated than with the purely rule-based systems.

The three examples Rolling Balls, Game of Life, and Melting Ice illustrated how Cartoonist deals with a wide range of parallel behaviors. In Chapter 4 this will be discussed in detail and it will be shown that Cartoonist can specify more parallel behaviors than the existing rule-based systems.

# Chapter 4

## Parallel Actions

This chapter discusses the characteristics and problems of rule-based systems that execute several actions in parallel. A framework is introduced in which a wide range of sequential and parallel rule-based systems can be described, including Cartoonist. Using this framework, the different rule-based systems are compared to Cartoonist and it is shown that Cartoonist has a greater functionality than the other systems. Finally, the time complexity of Cartoonist is compared to that of parallel rule-based systems.

### 4.1 Parallelism in Rule-Based Systems

This section introduces some of the main issues for rule-based systems that execute actions in parallel. Most of the existing rule-based systems execute actions in parallel to increase the system's efficiency. However, visual rule-based systems that are used to describe simulations often need to execute actions in parallel to accomplish the correct behavior of the objects in the simulation. For instance, if in a simulation, ice does not melt in parallel at the surface, it takes as much time to melt an ice block as it takes to melt the same amount of crushed ice, which is physically incorrect.

#### 4.1.1 Executing Actions in Parallel

Rule-based programs are generally sequential in the sense that they execute one action per cycle. Increasing the efficiency of a rule-based system and increasing the expressiveness of the language of rule-based systems are reasons to execute more than one action in parallel. Some of the problems connected with executing actions in parallel are discussed next. Schmolze's framework is used in a slightly modified form [68].

A rule  $r_i$  is of the form  $c_i \rightarrow a_i$ , where  $c_i$  is the condition and  $a_i$  is the action. A rule condition  $c_i$  matches state  $s$  if  $c_i$  is true in  $s$ . An action adds facts to a state and removes

acts from it. A condition and an action can be partitioned into a positive and a negative part, such that,  $c_i = c_i^+ \cup c_i^-$  and  $a_i = a_i^+ \cup a_i^-$ .

**MATCH:** A condition  $c$  matches state  $s$ , written  $c(s)$ , if there is a variable substitution  $\sigma$ , such that

$$c(s) \equiv \exists \sigma. c^{+\sigma} \subseteq s \wedge c^{-\sigma} \cap = \emptyset$$

In other words,  $c(s)$  holds if state  $s$  is consistent with condition  $c$ . When  $c(s)$  is true, then the action associated with this match is  $a^\sigma = a^{+\sigma} \cup a^{-\sigma}$ , which applied to  $s$  results in  $s'$ , that is  $s' = a^\sigma(s)$ , where

$$a(s) \equiv (s \setminus a^{-\sigma}) \cup a^{+\sigma} = (s \cap \overline{a^{-\sigma}}) \cup a^{+\sigma}$$

where  $x \setminus y$  is the difference of the two sets  $x$  and  $y$ ; that is,  $x \setminus y$  contains all those elements in  $x$  but not in  $y$ .

In sequential rule-based systems, the following cycle of operations is executed repeatedly.

- **Match** all rules against the current state forming the *conflict set of rule instances*.
- **Select** one rule instance from the conflict set according to the *conflict resolution strategy*.
- **Fire** the selected instance, that is, apply the action to the current state resulting in a new state.

Parallelizing this algorithm by simply extending the select- and the fire-phase to the whole conflict set does not work in general because of unintended possible interactions between the actions. If the parallel execution of the actions should lead to the same result as some sequential execution does, then the set of actions executed in parallel must be further screened. A set of actions can be executed if the rule instances associated with the actions are independent.

**INDEPENDENCE OF RULE INSTANCES:** Two rule instances  $r_i$  and  $r_j$  are independent if they neither *clash* nor *disable* each other. An instance  $r_i$  disables an instance  $r_j$  if  $r_i$  adds a fact that  $r_j$  tests for its absence or if  $r_i$  removes a fact that  $r_j$  tests for its presence, that is, if

$$a_i^+ \cap c_j^- \cup a_i^- \cap c_j^+ \neq \emptyset$$

Two instances clash if  $r_i$  adds a fact that  $r_j$  removes or vice versa, that is, if

$$a_i^+ \cap a_j^- \cup a_i^- \cap a_j^+ \neq \emptyset$$

This definition implies that the result is independent of the application's order of two independent rule instances. The result is the same as if the two instances had been applied in parallel, that is,

Assume instance action  $a_1$  is applied to  $s$  and then action  $a_2$  to  $a_1(s)$ .

$$a_2(a_1(s)) = (((s \setminus a_1^-) \cup a_1^+) \setminus a_2^-) \cup a_2^+ = (((s \cap \overline{a_1^-}) \cup a_1^+) \cap \overline{a_2^-}) \cup a_2^+$$

Reordering this expression leads to

$$a_2(a_1(s)) = (s \cup a_1^+ \cup a_2^+) \cap (\overline{a_1^-} \cup a_1^+ \cup a_2^+) \cap (\overline{a_2^-} \cup a_2^+)$$

$\overline{a_i^-} \cup a_i^+ = \overline{a_i^-}$  holds because  $a_i^+ \cap a_i^-$  is true. Therefore, the previous expression can be simplified to

$$a_2(a_1(s)) = (s \cup a_1^+ \cup a_2^+) \cap (\overline{a_1^-} \cup a_2^+) \cap \overline{a_2^-}$$

For the same reasons  $a_i^+ \cap \overline{a_i^-} = a_i^+$  holds. Therefore,

$$a_2(a_1(s)) = (s \cup a_1^+ \cup a_2^+) \cap ((\overline{a_1^-} \cap \overline{a_2^-}) \cup a_2^+) = (s \cup a_1^+ \cup a_2^+) \cap ((\overline{a_1^-} \cup \overline{a_2^-}) \cup a_2^+)$$

This expression can be rewritten as

$$a_2(a_1(s)) = ((s \cup a_1^+ \cup a_2^+) \cap \overline{a_1^- \cup a_2^-}) \cup a_1^+$$

Because clashes have to be avoided, that is, because  $a_1^- \cap a_2^+ = \emptyset$ , the previous expression can be further simplified, resulting in

$$a_2(a_1(s)) = (s \cup a_1^+ \cup a_2^+) \cap \overline{a_1^- \cup a_2^-} = (s \setminus (a_1^- \cup a_2^-)) \cup (a_1^+ \cup a_2^+)$$

The result is symmetric, so

$$a_2(a_1(s)) = a_1(a_2(s))$$

Furthermore,  $(s \setminus (a_1^- \cup a_2^-)) \cup (a_1^+ \cup a_2^+)$  is also the result if the two actions are applied in parallel. To show that any sequential order of application of independent rule instances leads to the same result as their parallel application, only the clashing condition was used in above proof. It was assumed that the actions could be fired in two different orders, which can happen only if the rules do not disable each other. This result suggests the following parallel rule-based system.



- **Match** all rules against the current state forming the conflict set of rule instances.
- **Select** repeatedly a rule instance from the conflict set according to a conflict resolution strategy and add it to a set of parallel actions if the new action does not clash with any action in the set of parallel actions.
- **Fire** all the instances in the set of parallel actions; that is, apply the actions in parallel to the current state, resulting in a new state.

### 4.1.2 Increasing Efficiency

A reason why the parallel approach may improve the efficiency of the rule-based system is that the execution of many actions leads to more changes in the state per cycle, and thus the conflict set contains more rule instances in the next cycle.

Approaches similar to the parallel match-select-fire cycle are commonly used to increase efficiency as well as to describe parallel processes with rule-based systems in visual programming environments. However, if the goal of executing many actions in parallel is to increase efficiency, then it must be insured that the rules do not disable each other.

For instance, the execution model in the CREL (Concurrent Rule Execution Language) system [42] defines the correctness of a CREL program as follows. In each cycle  $E_i$  actions  $a_{i_1}, \dots, a_{i_m}$  are executed in parallel.  $E_i$  is serializable if there is a serial execution  $E'_i$  of  $E_i$  such that  $E'_i$  produces the same result as  $E_i$ . A CREL program is correct if all serial executions reach correct terminal states. This means that any sequentialization of the parallel program leads to a correct result. Other parallel rule-based systems use similar definitions of a correct parallel execution of a rule-based program [38, 74].

There are different ways to accomplish correct parallel programs. The detection of the parallelism is *implicit* or *explicit* [34]. If the parallelism is implicit, then the rule-based system has to analyze the rules and rule instances to decide which actions can be executed in parallel. Some analysis can be done during compile time; however, whether two actions can be coexecuted often depends on the variable bindings of the actions [3, 38, 41]. A higher parallelism requires a more advanced analysis, which results in an increased overhead.

Explicit detection of parallelism shifts the task of analyzing the program to the programmer. The rules can be statically decomposed into modules to take advantage of the inherent parallelism in the problem domain [34]. Another approach is to use meta-rules to determine which rules may be applied in parallel.

The PARULEL language uses meta-rules to describe the parallelism in the program at the algorithmic level [74]. The meta-rules implement the conflict resolution strategy used

in the selection phase of the parallel version of the match-select-fire cycle. The programmer must define which actions may fire in parallel, and the system does no further tests. As the name implies, these meta-rules are about the task level rules; that is, they do not refer to the problem domain.

### 4.1.3 Increasing Expressiveness

Parallelism can increase expressiveness because a purely sequential language cannot always describe what the parallel language can. For instance, in the Game of Life actions may disable other actions which results in the wrong behavior if the actions are executed sequentially.

If parallelism is used in a simulation only to make things look like they happened more or less in parallel, then the same approach as for improving efficiency can be used. In this case, actions that are executed in parallel are not required to happen in parallel, but it looks more natural to the observer. However, quite often, this kind of parallelism is not sufficient.

In a simulation, certain actions must happen in parallel, for instance when ice melts or a ball chain is rolling down a slope. If these actions are not executed in parallel, the resulting behavior is not correct. Different approaches have been tried to solve this problem in rule-based visual programming systems.

- In the KidSim version [72] presented at the Child's Play workshop, the problem is completely ignored and all the actions are executed sequentially. Inherently parallel actions cannot be programmed in KidSim, although approximations are possible.
- Use a simple match-select-“fire if it still matches” strategy that works in interesting cases as is done in ScienceShows. This approach is better than ignoring the problem completely; however, the class of programmable parallel tasks and thus the match-select-fire strategy is chosen rather arbitrarily.
- Agentsheets [63] provides the user with the possibility to change the match-select-fire cycle. A larger class of parallel problems than in ScienceShows is solvable, yet it is not so clear whether an end user is able to select the appropriate strategy. How one would deal with the situation in which different parts of the simulation needed a different strategy is unclear. This approach is an extension to the one chosen in ScienceShows, which might take a similar approach in the future.
- Meta-rules can be used to describe which rules are to be executed in parallel, as is done in the non-visual PARULEL [74]. These rules implement the conflict-resolution strategy that chooses the actions to be executed in parallel. This approach is more

```

EXECUTE(s, Select, Applicable, Done)
  repeat
    Display(s)
     $s' \leftarrow s$ 
     $s \leftarrow \text{COEXECUTE} (s, \text{Select}, \text{Applicable}, \text{Done})$ 
  until  $s = s'$ 
end EXECUTE

```

**Figure 4.1:** Main loop of the rule-based system.

flexible and extendible than having built-in strategies as Agentsheets has; however, it is also much harder to write these meta-rules than just choosing one of the provided strategies.

- In Cartoonist, the user assumes a set of rules and programs the conflict-resolution strategy with cartoons. Although the rule-like cartoons have the same function as PARULEL's meta-rules, they are not about rules but about the task domain. Thus, they are not more difficult to write than the “normal” rules, although they are much more powerful.

To compare all these different approaches, a framework is required in which all the systems discussed can be expressed. Only then it is possible to make claims that one approach is better than another.

## 4.2 General Framework for Rule-Based Systems

A rule-based system can choose which action(s) to execute in many different ways. The previous sections showed some of the possibilities and presented the problems associated with the execution of several actions in parallel. In this section, a framework for rule-based systems is introduced in which all the sequential and parallel systems can be described. The framework is basically a more flexible version of the match-select-fire algorithm and consists of two parts. The main part, EXECUTE, shown in Figure 4.1, calls COEXECUTE repeatedly.

EXECUTE repeatedly displays the state and calls COEXECUTE (see Figure 4.2) until the new state returned by COEXECUTE has not changed. Then, the program execution stops. The invisible states are those which are generated in the for-loop in COEXECUTE but are not returned by this function. Thus, rule-based systems that change  $s$  in COEXECUTE once only, have no invisible states.

```

COEXECUTE(s, Select, Applicable, Done)
  repeat
     $A \leftarrow \text{Select}(\text{Match}(s))$ 
    for  $c \rightarrow a \in A$  do
      if Applicable( $c \rightarrow a, s$ ) then  $s \leftarrow a(s)$ 
    until Done( $A$ )
  return s
end COEXECUTE

```

**Figure 4.2:** COEXECUTE, the heart of the framework corresponds to the match-select-fire cycle in the simple framework.

```

COEXECUTE/1(s, CRS, true, true)
   $\{c \rightarrow a\} \leftarrow \text{CRS}(\text{Match}(s))$ 
   $s \leftarrow a(s)$ 
  return s
end COEXECUTE/1

```

**Figure 4.3:** COEXECUTE/1 is the description of the conventional sequential rule-based system. CRS returns a singleton containing the only action to be executed.

The function Match used in COEXECUTE returns a list of condition-action pairs written as  $c \rightarrow a$ . It is dependent on what kind of relations are allowed in the language used to describe the pictures in the rewrite rules and the cartoons.

COEXECUTE is parameterized with three functions that depend on what kind of rule-based system is being described. The conventional sequential rule-based system is described by the following definitions of the three functions.

Select( $X$ )	$\equiv$	conflict resolution strategy
Applicable( $c \rightarrow a, s$ )	$\equiv$	<i>true</i>
Done( $X$ )	$\equiv$	<i>true</i>

The function Select return a list of condition-action pairs. In this case, COEXECUTE is reduced to the function in Figure 4.3, where CRS is the conflict resolution strategy. Both functions Applicable and Done are set to *true* because the action selected by CRS will always be applicable and there is only one action in  $A$ , so a loop over  $A$  would keep repeating, applying the same action.

```

COEXECUTE/2(s)
   $c \rightarrow a \leftarrow \text{first}(\text{Match}(s))$ 
   $s \leftarrow a(s)$ 
  return s
end COEXECUTE/2

```

**Figure 4.4:** The first action instance is executed as in Agentsheets, BitPict, ChemTrains, and KidSim.

```

COEXECUTE/3(s)
   $c \rightarrow a \leftarrow \text{rand}(\text{Match}(s))$ 
   $s \leftarrow a(s)$ 
  return s
end COEXECUTE/3

```

**Figure 4.5:** A randomly chosen action instance is executed as it is possible in ChemTrains, ScienceShows, and KidSim.

### 4.2.1 Sequential Systems

The generic sequential rule-based system, introduced in Figure 4.3 as function COEXECUTE/1, can now be specialized for the rule-based visual systems Agentsheets, BitPict [28], ChemTrains, and KidSim, which all fire only one rule per cycle. As mentioned earlier, Agentsheets is normally programmed in CommonLisp. Whenever Agentsheets is referred to as a rule-based system, the rule-based extension of Agentsheets is meant. Most of these systems execute the first action that applies; that is, the conflict resolution strategy orders the rule instances, for example, according to the priority of the rules and picks the instance with the highest priority. The parameter functions are then defined as follows

$$\begin{aligned}
 \text{Select}(X) &\equiv \{\text{first}(X)\} \\
 \text{Applicable}(c \rightarrow a, s) &\equiv \text{true} \\
 \text{Done}(X) &\equiv \text{true}
 \end{aligned}$$

Function  $\text{first}(X)$  returns the first element in  $X$ . The resulting COEXECUTE function is shown in Figure 4.4.

Two of the systems, ChemTrains and KidSim, also allow the user to specify that the system has to choose randomly between instances of certain rules. The only difference from the previous description is that  $\text{Select}(X)$  is defined to be  $\text{rand}(X)$ , where  $\text{rand}(X)$  returns an arbitrary element in  $X$  (see Figure 4.5). Because not just the end state of a

```

COEXECUTE/4(s)
  A ← Match(s)
  for c→a ∈ A do s ← a(s)
  return s
end COEXECUTE/4

```

**Figure 4.6:** The simple parallel algorithm used by Agentsheets can also be run in a different mode, as shown in Figure 4.7.

program is interesting when describing a simulation—if one exists at all—, but also the process, the conflict resolution strategy cannot be made too smart or the user might lose the necessary control over the program’s behavior. Soar, using a complicated preference scheme to implement the conflict resolution strategy, suggests this [45].

### 4.2.2 Parallel Systems

Sequential systems are conceptually simple, yet quite powerful. Parallel systems tend to be more complicated, although ScienceShows, for instance, is still quite simple. Cartoonist, on the other hand, pays with a rather complicated and computationally expensive algorithm for increased expressiveness.

The first system considered is the simplest parallel version, which assumes that all actions in the conflict set can be executed in parallel. This makes sense for certain examples, for instance, the Game of Life. As described in Section 3.4.2, the Game of Life is simulated on a grid in which each cell is either alive or dead. First, each cell must be tested as to whether it will be dead or alive in the next generation before the actual changes can be made because otherwise the cells’ neighborhoods would change and the behavior of the cells would not follow the rules anymore. The parameters for COEXECUTE shown below result in the simple function COEXECUTE/4 in Figure 4.6.

$$\begin{aligned}
 \text{Select}(X) &\equiv X \\
 \text{Applicable}(c \rightarrow a, s) &\equiv \text{true} \\
 \text{Done}(X) &\equiv \text{true}
 \end{aligned}$$

The Game of Life is a somewhat special case in that there is no harmful interaction between the actions. The rules describing the process are context-free in the language of map L-systems [62]. A simple way to deal with the cases where coexecuted actions can interact in a way that leads to the wrong outcome is to test the condition of the action again just before applying the action. The conflict set serves as a pool of actions that can be

```

COEXECUTE/5(s)
  A ← Select(Match(s))
  for c→a ∈ A do
    if c(s) then s ← a(s)
  return s
end COEXECUTE/5

```

**Figure 4.7:** The condition is checked again before the action is executed. This approach takes care of some of the disabling problems, but clashing is ignored. ScienceShows uses this algorithm which is also an option in Agentsheets.

executed in parallel. However, since an action might disable another action, the condition of the action is checked again. This approach, shown in Figure 4.7, requires us to define  $\text{Applicable} \equiv c(s)$ . The strategy is adopted by ScienceShows and is available as an option in Agentsheets. It does not deal however with the problem of clashing, which means that, depending on how the actions are ordered, a different end state may be reached. The melting ice example, for instance, works properly with COEXECUTE/4.

### 4.3 Cartoonist Approach

The parallel approaches either are too hard to use or can solve only a relatively small subset of all of the interesting parallel problems. Providing a set of algorithms from which the user can choose can also be problematic because it requires the end user to make difficult decisions in the computational domain in which the user is not supposed to have any expertise. The same is true if meta-rules have to be used to specify the kind of parallelism necessary. On the other hand, meta-rules are at least more flexible than built-in strategies.

Cartoonist introduces the possibility of programming the conflict resolution strategy with a mechanism which is similar to meta-rules, yet to the user the mechanism looks much more like rules in the task domain. The definition of the parameter functions is given below, where  $c \rightarrow a \succ c' \rightarrow a'$  is true in state  $s$  if the best supporter of  $a(s)$  is preferred over the best supporter of  $a'(s)$ .

$$\begin{aligned}
 \text{Select}(X) &\equiv \{\text{rand}(\{c \rightarrow a \mid c \rightarrow a \in X \wedge \\
 &\quad \neg \exists c' \rightarrow a' \in X. c' \rightarrow a' \succ c \rightarrow a\})\} \\
 \text{Applicable}(c \rightarrow a, s) &\equiv \text{true} \\
 \text{Done}(X) &\equiv X = \emptyset
 \end{aligned}$$

```

COEXECUTE/7A(s)
  repeat
    A ← Match(s)
    R ← ∅
    for c→a ∈ A do
      s' ← a(s)
      R ← R ∪ {⟨s'; v1, ..., vk⟩}, where v1, ..., vk are
                                     all the voters for s'
    end for
    S ← {s' | r = ⟨s'; v1, ..., vk⟩ ∧ r ∈ R ∧ ¬∃r'. r' ∈ R ∧ r' ⤵ r}
    s ← rand(S)
  until S = ∅
  return s
end COEXECUTE/7A

```

**Figure 4.8:** This parallel system is used in Cartoonist. The user mainly programs the cartoons that generate the voters and not the rules generating the conflict set  $A$ .

Select( $X$ ) returns the set of best  $c \rightarrow a$  pairs. The resulting COEXECUTE function is shown in Figure 4.8. This version can be further simplified by taking advantage of the SUBSUMPTION function defined in Figure A.8. SUBSUMPTION computes exactly what is needed in COEXECUTE, as shown below.

$$\text{SUBSUMPTION}(s, A) = \{s' \mid c \rightarrow a \in A \wedge s' = a(s) \wedge \neg \exists c' \rightarrow a' \in A. c' \rightarrow a' \succ c \rightarrow a\}$$

The algorithm COEXECUTE/7A in Figure 4.8 shows that Cartoonist is indeed an instance of the framework introduced in this chapter, whereas the algorithm COEXECUTE/7B in Figure 4.11 is better suited to explain how Cartoonist works and how central the idea of subsumption is in Cartoonist. It also shows that Cartoonist's functionality is a superset of the other rule-based systems' functionality. A more compact way of describing the same algorithm used in describing CARTOONIST is the complete algorithm in the Cartoonist engine shown in Figure 4.9.

### 4.3.1 Other Parallel Behavior

There are problems that are not covered by any of the execution strategies discussed so far. The *Frogs and Flies* simulation consists of flies and frogs that eat these flies. In one time step (or cycle) as many flies as possible should be eaten, but no frog can eat more than one fly at once, nor can the same fly be eaten by more than one frog. The last requirement is



```

CARTOONIST(s)
  repeat
    Display(s)
     $s' \leftarrow s$ 
    loop
       $A \leftarrow \text{Match}(s)$ 
       $S \leftarrow \text{SUBSUMPTION}(s, A)$ 
      if  $S = \emptyset$  then exit loop
       $s \leftarrow \text{rand}(S)$ 
    end loop
  until  $s = s'$ 
end CARTOONIST

```

**Figure 4.9:** The complete algorithm used in Cartoonist combines EXECUTE and COEXECUTE.

rather obvious, however, it makes the problem hard, because without this requirement, the solution to the melting ice problem from Chapter 3 could be used.

Unfortunately, the frogs and flies problem cannot be dealt with any of the parallel algorithms described in the general framework including Cartoonist. Although the existing parallel systems could be extended to solve the Frogs and Flies problem, it is questionable whether these add-hoc solutions are useful for a wide range of examples. Furthermore, it complicates the use of the systems because the user is required to make the decision which kind of execution strategy to use.

Assume that COEXECUTE/5 in Figure 4.7 is used to attempt the frogs and flies problem. The conflict set  $A$  contains all the actions by which a frog can eat a fly. Generally, some frogs and some flies are mentioned more than once, however, the frogs and flies task description requires that no frog nor fly be mentioned more than once in the set of coexecuted actions. Two solutions are obvious.

Define  $\text{Applicable}(c \rightarrow a, s)$  such that it never executes an action if it contains objects that have been mentioned before in a conditional action. The following definitions accomplish this, resulting in function COEXECUTE/6 displayed in Figure 4.10.

```

Select( $X$ )            $\equiv X$ 
Applicable( $c \rightarrow a, s$ )  $\equiv$  if  $C \cap (c \cup a) = \emptyset$  then  $C \leftarrow C \cup c \cup a$ ; true else false end if
Done( $X$ )              $\equiv true$ 

```

The set  $C$  keeps track of what has been mentioned in the conditional actions. An action can be executed only if parts of the conditional action are not already in  $C$ . Although

```

COEXECUTE/6(s)
  C ← ∅
  repeat
    A ← Match(s)
    for c→a ∈ A do
      if C ∩ (c ∪ a) = ∅ then
        C ← C ∪ c ∪ a; s ← a(s)
    until Done(A)
  return s
end COEXECUTE/6

```

**Figure 4.10:** This parallel algorithm solves the frogs and flies problem.

this solution works for the frogs and flies simulation, it is questionable whether it is general enough to be of any other practical use.

Another way to solve the frogs and flies problem is to use meta-rules as in PARULEL, which filter the conflict set. We could then use the default strategy of ScienceShows (see Figure 4.7) and add a meta-rule that makes sure no frog is mentioned more than once. There is no need to do the same for the flies because they are eaten, and thus the condition is not satisfied anymore when the action is to be executed. This approach is not satisfying either, because the intended user of the system is an end user who is not interested in writing rules about rules just to make the frogs eat properly.

### 4.3.2 Comparing the Different Approaches

All the approaches fit into a general framework. Table 4.1 summarizes the different algorithms, ignoring the initialization of variables.

The Cartoonist version of the COEXECUTE algorithm is described in Figure 4.11 using an incremental version of matching the modified state *s*. This makes it easier to show how COEXECUTE/7B compares with the existing sequential and especially parallel rule-based systems. Disable(*a*, *s*) and Enable(*a*, *s*) have to be defined such that the line  $A \leftarrow (A \setminus \text{Disable}(a, s)) \cup \text{Enable}(a, s)$  could be replaced by  $s \leftarrow \text{Match}(s)$ ; that is, the following relationships must hold:

$$\begin{aligned}
 A &= \text{Match}(s) \\
 (A \setminus (\text{Disable}(a, s)) \cup \text{Enable}(a, s)) &= \text{Match}(a(s))
 \end{aligned}$$

**Table 4.1:** Summary of all the different instantiations of COEXECUTE. Footnote marks in parentheses mean that the system might have this functionality at some time in the future.

	# <sup>1</sup>	Select( $X$ )	Applicable( $c \rightarrow a, s$ )	Done( $X$ )
generic system	1	conflict resolution	<i>true</i>	<i>true</i>
first action <sup>abde</sup>	2	{first( $X$ )}	<i>true</i>	<i>true</i>
random action <sup>def</sup>	3	{rand( $X$ )}	<i>true</i>	<i>true</i>
all at once <sup>(a)</sup>	4	$X$	<i>true</i>	<i>true</i>
test again <sup>(a)f</sup>	5	$X$	$c(s)$	<i>true</i>
part. max once	6	$X$	$f_1(c)$	<i>true</i>
sequential parallelism <sup>c</sup>	7	{rand( $f_2(X)$ )}	<i>true</i>	$X = \emptyset$

$f_1(c) = \text{if } C \cap c = \emptyset \text{ then } C \leftarrow C \cup c; \text{ true else false end if}$

$f_2(X) = \{c \rightarrow a \mid c \rightarrow a \in X \wedge \neg \exists c' \rightarrow a' \in X. c' \rightarrow a' \succ c \rightarrow a\}$

<sup>1</sup># is the index in COEXECUTE/#

<sup>a</sup>Agentsheets, with rule extension

<sup>b</sup>BitPict

<sup>c</sup>Cartoonist

<sup>d</sup>ChemTrains

<sup>e</sup>KidSim

<sup>f</sup>ScienceShows

Therefore, Disable( $a, s$ ) and Enable( $a, s$ ) can be defined as:

$$\text{Disable}(a, s) \equiv \text{Match}(s) \setminus \text{Match}(a(s))$$

$$\text{Enable}(a, s) \equiv \text{Match}(a(s)) \setminus \text{Match}(s)$$

Disable( $a, s$ ) contains the conditional actions that are *disabled* by applying  $a$  to  $s$  and Enable( $a, s$ ) contains the conditional actions that are *enabled* when  $a$  is executed in  $s$ . By constraining Disable and Enable to certain values, COEXECUTE/7B can simulate other sequential and parallel COEXECUTE algorithms.

Each type of rule-based system in Table 4.1 will be compared to the Cartoonist system. For each existing system, it will emerge that Cartoonist has at least the same functionality. The algorithm COEXECUTE/6, which cannot be emulated by Cartoonist, is not employed by any system. Finally, because Cartoonist can describe programs that none of the other systems can, Cartoonist has a greater functionality than all the other visual rule-based systems considered in this chapter. Although Cartoonist can simulate a sequential rule-based system as is shown later, it is not clear that there are examples in which a sequential execution of the rules is preferred over a parallel one.

```

COEXECUTE/7B(s)
  A ← Match(s)
  while A ≠ ∅ do
    a ← rand({a | a(s) ∈ SUBSUMPTION(s)})
    A ← (A \ Disable(a, s)) ∪ Enable(a, s)
    s ← a(s)
  end while
end COEXECUTE/7B

```

**Figure 4.11:** This algorithm is used in Cartoonist and is the same as in Figure 4.8. The explicit use of the functions  $\text{Disable}(s)$  and  $\text{Enable}(s)$  makes it easier to compare the cartoon approach with the ones discussed earlier.

```

COEXECUTE/7B/SEQ(s)
  A ← Match(s)
  a ← rand({a | a(s) ∈ SUBSUMPTION(s)})
  s ← a(s)
end COEXECUTE/7B/SEQ

```

**Figure 4.12:** The Cartoonist algorithm with  $\text{Disable}(a, s) = A$  and  $\text{Enable}(a, s) = \emptyset$  becomes a sequential algorithm.

#### 4.3.2.1 Sequential: Random Action

Suppose the constraints on  $\text{Disable}(a, s)$  and  $\text{Enable}(a, s)$  are such that only one action can be executed; that is, suppose all actions must be disabled after executing the first action and no new action must be enabled.

$$\text{Disable}(a, s) = A$$

$$\text{Enable}(a, s) = \emptyset$$

These conditions reduce  $\text{COEXECUTE/7B}$  to a sequential algorithm in Figure 4.12.

It has to be shown that the correct choice of rules and cartoons leads to  $\text{Disable}(a, s)$  and  $\text{Enable}(a, s)$ , as required above. Let  $\mathcal{R}$  be the rules in a sequential system in which one rule is fired, either the first in the conflict set or an arbitrary one from the set, depending on the system's strategy.

The following notation of rules and cartoons is assumed: A rule is of the form  $(c^+, c^-) \rightarrow (a^+, a^-)$ , meaning that the condition  $(c^+, c^-)$  and the action  $(a^+, a^-)$  consist of a positive and a nega-

tive part each; similarly for a constraint  $(C_1^+, C_1^-) \Rightarrow (C_2^+, C_2^-)$  with the constraints  $C_1$  and  $C_2$ . The notation  $\Rightarrow$  is used for cartoons to differentiate them from rules.

Given the set of rules  $\mathcal{R}$  of a sequential rule-based system, a set  $\mathcal{R}'$  of rules and a set  $\mathcal{C}$  of cartoons must be found such that the conditions for  $\text{Disable}(a, s)$  and  $\text{Enable}(a, s)$  hold and the correct strategy is used to select the only action to execute.

$\mathcal{R}'$  is constructed so that only one action can be executed in parallel. This is done by introducing the boolean flag  $\lambda$ . All the rules check this fact and change it, that is, that the rules toggle  $\lambda$ . The rule set  $\mathcal{R}'$  is then defined as follows.

$$\mathcal{R}' \equiv \{(c^+ \cup \{\lambda\}, c^-) \rightarrow (a^+, a^- \cup \{\lambda\}) \mid (c^+, c^-) \rightarrow (a^+, a^-) \in \mathcal{R}\} \cup \{(c^+, c^- \cup \{\lambda\}) \rightarrow (a^+ \cup \{\lambda\}, a^-) \mid (c^+, c^-) \rightarrow (a^+, a^-) \in \mathcal{R}\}$$

$\mathcal{C}$  contains only two cartoons that describe how  $\lambda$  switches from being true to being false to being true, and so on.

$$\mathcal{C} \equiv \{(\{\lambda\}, \{\}) \Rightarrow (\{\}, \{\lambda\}), (\{\}, \{\lambda\}) \Rightarrow (\{\lambda\}, \{\})\}$$

or, in a bit more readable format

$$\mathcal{C} \equiv \{\{\lambda\} \Rightarrow \{\neg\lambda\}, \{\neg\lambda\} \Rightarrow \{\lambda\}, \}$$

The Cartoonist system with the rules  $\mathcal{R}'$  and the cartoons  $\mathcal{C}$  executes never more than one action in parallel, that is, the actions are executed sequentially. This can be seen as follows.

Assume that  $\lambda$  is true and the conflict set  $A$  is not empty. Because each action in  $A$  sets  $\lambda$  to false, the first cartoon describes this state sequence. Any second action that would be executed would set  $\lambda$  to true again. However, there is no cartoon that describes a state sequence where the truth of  $\lambda$  does not change. Therefore, at most only one action in  $A$  can be executed. The same reasoning applies when  $\lambda$  is initially false.

Using the  $\mathcal{C}$  with two cartoons implements the *random action* conflict resolution strategy.

#### 4.3.2.2 Sequential: First Action

The strategy in which the first action is always chosen requires the following, based on the random strategy above.  $\mathcal{C}$  needs to be extended so that the first action gets the best support. This can be accomplished with following definition.

$$\mathcal{C} \equiv \{\{\lambda\} \Rightarrow \{\neg\lambda\}, \{\neg\lambda\} \Rightarrow \{\lambda\}, \} \cup \{c \Rightarrow a \mid c \rightarrow a \in \mathcal{R}\}$$

A rule  $r_i$  coming before another rule  $r_j$  in the sequential system, that is,  $r_i, r_j \in \mathcal{R}$ , is reflected in Cartoonist by the preference relation such that the cartoon corresponding to  $r_i$  is preferred over the cartoon corresponding to rule  $r_j$ .

This shows that Cartoonist can simulate the sequential rule-based systems based on COEXECUTE/2 and COEXECUTE/3.

### 4.3.2.3 Parallel: All Actions

Cartoonist can also simulate the case in which all the actions are executed in parallel without further testing of the condition as described by COEXECUTE/4. The conditions on  $\text{Disable}(a, s)$  and  $\text{Enable}(a, s)$  for this case are as follows.

$$\begin{aligned}\text{Disable}(a, s) &= \{a\} \\ \text{Enable}(a, s) &= \emptyset\end{aligned}$$

This means that every action executed is removed from the conflict set and no additional action is added to the set.

In a system that executes all actions in the conflict set  $A$  in parallel, the actions must be consistent in a domain-specific sense. Being clashfree is not good enough. For instance, if an action moves an object to the left and another action moves the same object to the right, then they do not clash. However, both actions must not be fired in parallel because an object cannot be moved to the right and left at once. This means that no two actions that contradict each other can be fired in parallel. Formally, the condition required for parallel application is

$$\forall a_i, a_j \in A. a_i^+ \cap a_j^- \cup a_i^- \cap a_j^+ = \emptyset$$

If this condition is not satisfied, the meaning of applying the actions in parallel is not clear. In the Game of Life, for instance, a cell is changed by exactly one action and no clash can happen.

Let the sets  $\mathcal{R}'$  and  $\mathcal{C}$  be defined as follows.

$$\begin{aligned}\mathcal{R}' &= \{(a^-, a^+) \rightarrow (a^+, a^-) \mid (c^+, c^-) \rightarrow (a^+, a^-) \in \mathcal{R}\} \\ \mathcal{C} &= \{c \Rightarrow a \mid c \rightarrow a \in \mathcal{R}\}\end{aligned}$$

The rules in  $\mathcal{R}'$  are *decontextualized* versions of the ones in  $\mathcal{R}$ . A decontextualized rule describes only the changes without any additional context, that is, only those facts are referenced in the action that are actually changed. Because the conditional actions in the conflict set  $A$  are consistent in the task domain, and the rules in  $\mathcal{R}'$  are decontextualized, they cannot disable each other.

However, the actions in  $A$  can enable other actions that are initially not in the conflict set. This is not a problem, because the cartoons enable only those actions that are initially in the conflict set. Furthermore, a cartoon always describes the shortest state sequence only and therefore,  $a \in \text{Disable}(a, s)$  for all actions  $a$  and states  $s$ .

#### 4.3.2.4 Parallel: Test Before Execute

Some parallel rule-based systems, notably ScienceShows, test the action again just before the action is executed, as is described by COEXECUTE/5. This case is even simpler, executing all actions in parallel. The rules need not to be decontextualized, that is,

$$\begin{aligned}\mathcal{R}' &= R \\ \mathcal{C} &= \{c \Rightarrow a \mid c \rightarrow a \in \mathcal{R}\}\end{aligned}$$

Again, the cartoons make sure that only those actions that were in the initial conflict set are executed.

#### 4.3.2.5 Parallel: Enable( $a, s$ ) $\neq \emptyset$

All the rule-based systems are in trouble if a set of parallel actions should be described where Enable( $a, s$ ) is not empty, that is, if the actions to be executed in parallel are not a subset of the actions in the conflict set. An example of such a case is the ball chain shown in Chapter 3. Cartoonist can deal with such cases, whereas all the earlier discussed rule-based systems cannot solve such problems. However, this does not imply that Cartoonist can deal with all problems, as the frogs and flies problem shows. Nevertheless, it shows that Cartoonist can describe a larger class of parallel actions than can the other rule-based systems. The rolling balls and the Pachinko examples in Chapter 2 show that Cartoonist solves a relevant set of additional problems.

### 4.3.3 Parallel Actions in Cartoonist

The previous section has shown that Cartoonist can simulate sequential systems and that it can describe a wider range of parallel actions than the other parallel rule-based systems can. To show that Cartoonist can simulate sequential systems required the introduction of a flag  $\lambda$  that is toggled by the rules. If this were necessary in practice, then it would be a serious problem because the necessary sets of rules and cartoons are quite large and not intuitive. Examples where a sequential application is necessary tend to have a flag already as part of the simulation. For example, in the game Tic Tac Toe the flag is meaningful as it represents who is to move next.

The situation is made favorable for parallel executions, in that Cartoonist supports a range of parallelism with no special tricks required. Three kinds of parallelisms have been discussed based on the examples of rolling balls, melting ice, and the Game of Life. The rule-based systems either solve the melting ice or the Game of Life, but not both. If they solve both, then the user needs to explicitly change the execution strategy as in Agentsheets.

```

Match(s)
repeat
  for  $i = 1, \dots, p$  do
    for  $c \rightarrow a \in \text{Match}(s)$  do
      Match( $a(s)$ )
    end for
  Match(s)
end for
until done

```

**Figure 4.13:** Reduced version of the Cartoonist engine algorithm from Figure 5.1. Only the expensive calls to the rule-base are retained.

In Cartoonist, nothing like that is necessary, although the solutions are just as natural as the ones in the rule-based systems.

In addition, Cartoonist solves some problems that the rule-based systems cannot, as the rolling balls show. In simulations, these kinds of situations occur often as can be seen in the PacMan example.

## 4.4 Time Complexity

Declarative programming languages tend to have less efficient implementations than procedural languages, although there are efficient Prolog compilers. Furthermore, rule-based systems tend to be relatively slow too due to the expensive matching phase [75].

As can be seen in Figure 5.1 and the definitions of  $\text{Match}(s)$ ,  $\text{CCI}(s)$ , and  $\text{GetVoters}(s)$ , it is only  $\text{Match}(s)$  that has to be taken into consideration to compute the time complexity of the Cartoonist engine. Although calculating  $R$  and  $S$  in Figure 5.1 cannot be done in constant time, this computation time can be neglected compared to the cost of  $\text{Match}(s)$ . Figure 4.13 shows the Cartoonist engine's algorithm in a form that makes it easier to assess its time complexity.

Let  $T_c(p)$  be the time it takes CARTOONIST to execute  $p$  actions in parallel. This is the same time as executing COEXECUTE/7 once. Executing these  $p$  actions means executing the for-loop in Figure 4.13  $p$  times. Let  $T(\text{Match})$  be the time to execute  $\text{Match}(s)$  and assume that there are at most  $b$  actions to be executed in a state, that is,  $|\text{Match}(s)| \leq b$ . Then  $T_c(p)$  is

$$T_c(p) = p \cdot (T(\text{Match}) + b \cdot T(\text{Match}) + T(\text{Match})) = p \cdot (b + 2) \cdot T(\text{Match})$$



The asymptotic upper bound of  $T(\text{Match})$  depends on what kinds of patterns are allowed in the rules' conditions and the cartoons' constraints. Because a Rete-like [26], and thus rather general pattern language is used,  $T(\text{Match})$  is exponential. The analysis of the matching cost is based on Tambe, Newell, and Rosenbloom's work on expensive chunks using the notion of a *k-search tree* [75]. The asymptotic upper bound of matching a rule-base with  $n$  rules is

$$O(\text{Match}) = n \cdot B^D,$$

where  $B$  is the branching factor of the  $k$ -search tree and  $D$  is its depth. The depth of the tree is dependent on the conditions of the rule and the branching factor is given by the number of the facts (working memory elements) that match the conditions.

$T_c(p)$  is compared with the simplest parallel rule-based system, COEXECUTE/4 shown in Figure 4.6. COEXECUTE/4 executes the  $p$  parallel actions without further testing. Thus, its time complexity  $T_p(p)$  is

$$T_p(p) = p \cdot T(\text{Match}')$$

$T(\text{Match}')$  is basically the same as  $T(\text{Match})$ ; however, the number of rules used by the two systems for the same problem may vary. In the worst case they are  $T(\text{Match}) = n_c \cdot B^D$  and  $T(\text{Match}') = n_p \cdot B^D$ , where  $n_c$  rules are in the Cartoonist's rule-base and  $n_p$  rules are in the parallel rule-based system's rule-base. A reasonable way to compare the time complexity of the two systems is by defining the factor  $f$  as follows.

$$f \equiv \frac{O(T_c)}{O(T_p)} = \frac{p \cdot (b+2) \cdot O(\text{Match})}{p \cdot O(\text{Match}')} = \frac{(b+2) \cdot n_c \cdot B^D}{n_p \cdot B^D} = (b+2) \cdot \frac{n_c}{n_p}$$

Thus,

$$f \approx b \cdot \frac{n_c}{n_p}$$

The factors  $b$  and  $n_c/n_p$  depend on the program implemented by the two systems. Nevertheless, the upper and lower bounds can be estimated. It will become clear shortly that the theoretical bounds are not of practical interest. Therefore, good estimates are of greater value than extreme theoretical bounds.

The number of actions  $b$  that can be executed in any situation can be estimated by  $o \cdot n_o$ , where  $o$  is the number of objects in the application and  $n_o$  is the average number of actions an object may execute. Moving balls have  $n_o \leq 4$ , in the melting ice example or  $n_o = 1$  in the Game of Life.

The value of  $b$  is due to the look-ahead search that is necessary because of the declarative semantics of the right-hand side of a cartoon. This big price that has to be paid is not intrinsic to the Cartoonist system but is caused by the implementation of testing whether a

cartoon can be satisfied. Furthermore, if  $n_c/n_p$  is small, that is, much smaller than 1, then there is the chance that a Cartoonist system is not necessarily more inefficient than a purely rule-based system.

I have shown that a Cartoonist system needs  $n_p$  rules and  $n_p$  cartoons to simulate the parallel rule-based system executing all actions in parallel without further testing. The factor  $f$  then becomes

$$f = o \cdot n_o \cdot \frac{n_p + 2 \cdot n_p}{n_p} = 3 \cdot o \cdot n_o$$

The factor 2 comes from the transformation of a cartoon with two constraints into two rules. For the *Game of Life*, this translates into  $f = 3 \cdot 1 \cdot n_{\text{cells}}$ ; that is, the Cartoonist version will be  $3 \cdot n_{\text{cells}}$  slower than the purely rule-based version, where  $n_{\text{cells}}$  is the number of cells.

This shows clearly where a major problem of cartoons and the current implementation is. However, the Game of Life or the melting ice examples can be viewed as worst-case scenarios, because these processes consists of a few huge interactions in Cartoonist.

In Section A.5.2, I have shown that  $m$  rules and  $n$  cartoons can require up to  $m \cdot 2^n$  rules in a purely rule-based system. This case, which cannot be expected in practice, leads to the theoretically best  $f$ .

$$f = o \cdot n_o \cdot \frac{m + l \cdot n}{m \cdot 2^n},$$

where  $n_o$  is the average number of constraints in a cartoon, which is about 3 for most applications encountered so far. A reasonable approximation for  $f$  can be obtained by approximating  $n_p$  with  $m \cdot n$ . By definition,  $n_c = m + l \cdot n$ , and therefore

$$f = o \cdot n_o \cdot \frac{m + l \cdot n}{m \cdot n}$$

This  $f$  looks better than the worst case, yet the fact that  $f$  is proportional to the number of objects  $o$  might suggest a more serious problem, namely that the system has difficulty in scaling up. The main reason for this problem lies in the completely global pattern-matching in the cartoons' constraints. This problem should be solvable, leading to a less global use of cartoons.

The global use of cartoons results in one big interaction per cycle, even if the characters do not interact with each other. This problem might be solved by somehow modularizing the interactions, that is, independent characters should not be meshed together into one huge interaction.



# Chapter 5

## Implementation of the Cartoonist Prototype

This chapter describes briefly the implementation of the Cartoonist prototype in Agentsheets. First, the complete Cartoonist algorithm is presented. Then, it is shown how cartoons can be translated into rules thus making the use of a quite conventional Rete-matcher possible. Finally, some issues to interface the Cartoonist engine with Agentsheets are discussed.

### 5.1 Cartoonist Algorithm

The main algorithm of the Cartoonist system is based on the two algorithms `CARTOONIST` and `SUBSUMPTION` developed in Chapter 4. For each action `Match` and `SUBSUMPTION` are executed once. In Figure 5.1, `SUBSUMPTION` is expanded and the two functions `ComputeCartoonInstances(s)` (CCI) and `GetVoters(s)` are introduced, which take care of all the necessary computations to find all of the cartoon instances and voters.

The most plausible idea to build the Cartoonist engine is to transform the cartoons into meta-rules implementing the conflict resolution strategy and use a rule-based system with a meta level. This approach would be similar to that taken in `PARULEL` [74], as described in Section 4.1.2 or various other sequential rule-based systems with a meta-level architecture [18, 25, 31].

However, cartoons are not at the meta-level because they do not say anything *about* rules. Cartoons are at the task level, like the rules. Nevertheless, they can take over the same task meta-rules normally do, that is, implementing which rule is to be fired next. In function `CARTOONIST`, the meta-component in a meta-level architecture would correspond to the `SUBSUMPTION` function.

```

CARTOONIST(s)
  ComputeCartoonInstances(s)
  repeat
    Display(s)
     $s' \leftarrow s$ 
    loop
       $A \leftarrow \text{Match}(s)$ 
      for  $c/a \in A$  do
         $s'' \leftarrow a(s)$ 
         $V \leftarrow \text{GetVoters}(s'')$ 
         $R \leftarrow R \cup \{ \langle s''; v_1, \dots, v_k \rangle \mid v_1, \dots, v_k \in V \wedge \forall i, i < k. v_i \succ v_{i+1} \}$ 
      end for
       $S \leftarrow \{ s \mid \langle s''; I \rangle = r \wedge \neg \exists r'. r' \in R \wedge r' \succ r \}$ 
      if  $S = \emptyset$  then exit loop
       $s \leftarrow \text{rand}(S)$ 
      ComputeCartoonInstances(s)
    end loop
  until  $s = s'$ 
end CARTOONIST

```

**Figure 5.1:** Result of expanding SUBSUMPTION in the CARTOONIST algorithm. The functions ComputeCartoonInstances(*s*) and GetVoters(*s*) that deal with the cartoons are defined later.

The fact that rules and cartoons are both at the task level suggests that an extension of a rule-based system could be the solution. Indeed, the prototype of Cartoonist is implemented using a temporal extension of a rule-based system, as will be shown in some detail in the next section.

The functions CCI(*s*) and GetVoters(*s*) are defined similarly to Match(*s*) with the exception that instead of returning conditional action pairs they return cartoon instances and voters, respectively. Voters are cartoon instances that cannot be further extended. Thus, CCI(*s*) and GetVoters(*s*) can be implemented as basically the same function.

Next I show how cartoons can be transformed into rules and used in a rule-based system. This makes it possible to estimate the time complexity of CARTOONIST and compare it to other rule-based systems.

$$\begin{array}{l}
R_1: \quad C_1, \text{current\_time}(t) \\
\quad \rightarrow \\
\quad (\alpha_1 \ t \ t \ \sigma_1) \\
\\
R_2: \quad (\alpha_1 \ t_0 \ t_1 \ \sigma_1) \\
\quad C_2, \text{current\_time}(t), \text{succ}(t_1, t) \\
\quad \rightarrow \\
\quad (\alpha_2 \ t_0 \ t \ \sigma_2) \\
\\
\quad \vdots \\
\\
R_i: \quad (\alpha_{i-1} \ t_0 \ t_{i-1} \ \sigma_{i-1}) \\
\quad C_i, \text{current\_time}(t), \text{succ}(t_{i-1}, t) \\
\quad \rightarrow \\
\quad (\alpha_i \ t_0 \ t \ \sigma_i) \\
\\
\quad \vdots \\
\\
R_k: \quad (\alpha_{k-1} \ t_0 \ t_{k-1} \ \sigma_{k-1}) \\
\quad C_k, \text{current\_time}(t), \text{succ}(t_{k-1}, t) \\
\quad \rightarrow \\
\quad (\alpha_k \ t_0 \ t \ \sigma_k) \\
\\
R_{k+1}: \quad (\alpha_k \ t_0 \ t_k \ \sigma_k) \\
\quad C_{k+1}, \text{current\_time}(t), t_k = t, \text{current\_state}(s) \\
\quad \rightarrow \\
\quad (\text{Vote } s \ \alpha \ t_0)
\end{array}$$

**Figure 5.2:** The rules resulting from the cartoon  $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$ .

## 5.2 Transforming Cartoons into Rules

Transforming a cartoon into rules, called c-rules to distinguish them from the “real” rules, is not as trivial as it might seem at first. It would simplify the implementation of the system a lot, if the c-rules could be used in the same rule-base, because rules and c-rules are at the task level and thus, match the same facts. Cartoons also have to match not just in one, but in several states and thus, have to be able to somehow refer to the past.

The cartoon  $\alpha = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$  is transformed into the rules shown in Figure 5.2. The condition  $C_1, \text{current\_time}(t)$  is the conjunction of the condition  $C_1$  and  $\text{current\_time}(t)$ , where the latter is always instantiated with the current time. The

fact  $\text{succ}(t', t)$  is true if  $t' + 1 = t$ .  $\text{Current\_state}(s)$  is also always instantiated with the current state. Finally,  $\sigma_i$  is a variable substitution containing all variable bindings of the variables in the previous constraints  $C_1, \dots, C_i$ , that is,  $\exists \Delta \sigma_i. \sigma_i = \sigma_{i-1} \cdot \Delta \sigma_i$  for all  $i$ ,  $1 \leq i < k + 1$ . All but the last rule generate a cartoon instance when fired. All but the first rule have to match the constraint  $C_i$  from the cartoon and the cartoon instance  $(\alpha_{i-1} \dots)$  that is extended when the rule is fired. The last rule adds a voter of the form

$$(\text{Vote } s \ \alpha \ t)$$

when fired. This voter votes for state  $s$ , the first state of the described state sequence was at time  $t$ , and the cartoon describing the state sequence is  $\alpha$ . This is more information than is actually needed, because the preference relation  $\succ$  for cartoons is only dependent on the static preference of  $\alpha$  (see page 110).

The functions  $\text{Match}(s)$ ,  $\text{CCI}(s)$ , and  $\text{GetVoters}(s)$  can now be defined in terms of  $M(s)$ , which returns the conditional actions as triples  $\langle c; d, a \rangle$ , where  $c$  is the condition,  $d$  are the facts to be removed from the working memory of the rule-based system, and  $a$  are the facts to be added to the working memory. Then, the three functions' definitions are as follows.

$$\begin{aligned} \text{Match}(s) &\equiv \{(a, d)/c \mid \langle c; d, a \rangle \in M(s) \wedge (\alpha_i \dots) \not\in d \wedge (\text{Vote} \dots) \not\in d\} \\ \text{CCI}(s) &\equiv \{a \mid \langle c; d, a \rangle \in M(s) \wedge (\alpha_i \dots) \in d\} \\ \text{GetVoters}(s) &\equiv \{a \mid \langle c; d, a \rangle \in M(s) \wedge (\text{Vote} \dots) \in d\} \end{aligned}$$

Thus, the Cartoonist engine can be implemented by a slightly extended *sequential* rule-based system. Although the cartoons have functions similar to those that as meta-rules have, no meta-level is required.

### 5.3 Interface between Cartoonist and Agentsheets

The interface between Agentsheets and Cartoonist is, conceptually, surprisingly simple. The characters are called *agents* in Agentsheets and they can be manipulated by sending them a message. The only three messages necessary are `CreateCharacter`, `DeleteCharacter`, and `ModifyCharacter`. `CreateCharacter` creates a character of a certain kind, for instance, a ball or PacMan, at a certain location. `DeleteCharacter` deletes a character and the most important method `ModifyCharacter` modifies a character's location.

In Cartoonist a character's status is represented by the fact

$$(\$character \langle type \rangle \langle id \rangle \langle x \rangle \langle y \rangle)$$

meaning that  $\langle id \rangle$  is the unique identifier of a character of type and depiction  $\langle type \rangle$  located at the coordinates  $(\langle x \rangle, \langle y \rangle)$ . At the beginning of a simulation, each agent in the simulation display reports its status to the Cartoonist engine, which builds the initial state from this

information. Cartoonist has a complete model of the simulation and does not need to get any more information from the Agentsheets part during the program execution.

Once Cartoonist is initialized, it finds the next state to be displayed, sends all the necessary methods to the agents and finds the next state, displays it, and so on until no new state can be found. The parallel execution of actions is accomplished as follows. Assume that the previously displayed state is  $s_0$  and that Cartoonist found the sequence  $s_1, \dots, s_n$  of states where the last state  $s_n$  is to be displayed again, that is, all the actions leading from  $s_0$  to  $s_n$  are executed at once, looking to the user like the parallel execution of  $n$  actions. The states  $s_0$  and  $s_n$  are real and the states  $s_1, \dots, s_{n-1}$  are virtual. Let  $a_i, i = 1, \dots, n$ , be the actions leading to state  $s_i$ , that is,  $s_i = a_i(s_{i-1})$ . Then, Cartoonist sends the methods translated from these actions to the Agentsheets component.

An action is in fact a conditional rewrite rule of the form  $c \mid b \rightarrow a$  and is translated into a method sent to an agent. The rule can be fired in  $s$  if there is a variable substitution  $\sigma$ , such that  $c^\sigma$  is true and  $b^\sigma \subseteq s$ . The rule is fired by replacing the before-picture  $b^\sigma$  with the after-picture  $a^\sigma$  in  $s$ . Thus, the execution of the methods sent to the agents must result in the new state computed by  $(s \setminus b^\sigma) \cup a^\sigma$ . The set of messages sent to the characters is computed by  $M(b^\sigma, a^\sigma)$ , which is defined as follows.

$$\begin{aligned}
 M(b, a) = & \{(\text{CreateCharacter } t \ x \ y) \mid \\
 & (\$character \ t \ i \ x \ y) \in a \wedge \neg \exists x', y'. (\$character \ t \ i \ x' \ y') \in b \} \cup \\
 & \{(\text{DeleteCharacter } i) \mid \\
 & (\$character \ t \ i \ x \ y) \in b \wedge \neg \exists x', y'. (\$character \ t \ i \ x' \ y') \in a \} \cup \\
 & \{(\text{ModifyCharacter } i \ x' \ y') \mid \\
 & (\$character \ t \ i \ x \ y) \in b \wedge (\$character \ t \ i \ x' \ y') \in a \}
 \end{aligned}$$

Given a rewrite rule  $b \rightarrow a$  and a variable binding  $\sigma$ , it is now easy to generate a set of messages  $M(b^\sigma, a^\sigma)$  to be sent to the agents.

Once the working memory of the Cartoonist part of the prototype is initialized, it completely takes over the simulation and decides what actions have to be executed in the simulation. Therefore, once the simulation is running, no interaction between the user and the simulation is possible. However, this is a problem of the ASC prototype and not of the Cartoonist framework.





# Chapter 6

## Conclusions

This last chapter first recapitulates how Cartoonist relates to some of the work mentioned in Chapter 1. Then, the chapter discusses how Cartoonist accomplishes the goals set out at the beginning of this dissertation. Finally it suggests some future work along the lines of the Cartoonist framework.

### 6.1 Related Work Revisited

This section discusses how Cartoonist relates to some of the issues introduced in Chapter 1.

#### 6.1.1 Rule-Based Languages

The actions of a character in Cartoonist are specified using OPS5-like rewrite rules. Each rule specifies one action and tends to be quite simple, for instance, moving to a free square. The user is required to program with rules only if the provided characters are not sufficient, either because a new character is needed or because the set of actions has to be changed.

Executing several rules in a sequence is supported by cartoons directly for some cases, as the ball that keeps moving in the same direction shows. Using a set of cartoons that can easily be generated by the system demonstrates this idea. Assume that we want the system to go through the states described by the constraints  $C_1, C_2, \dots, C_k$ . The set would then consist of the cartoons  $C_1 \rightarrow C_2, C_1 \rightarrow C_2 \rightarrow C_3, \dots, C_1 \rightarrow \dots \rightarrow C_k$ . This differs from having just the last cartoon, but may be an interesting extension.

#### 6.1.2 Declarative Languages

Cartoons are completely declarative because they only constrain the state sequences and do not describe how the states are generated. What makes a cartoon declarative is the way

the future state is described; the access to the past is irrelevant. Sometimes a cartoon looks like a procedural rule, however, this is only because it uses (sometimes) the same relations and predicates as the rules do. The declarative nature of the “right-hand side” of a cartoon makes it possible to use, for instance, negation to describe the next state. A rule cannot do that.

### 6.1.2.1 Logic Programming

Cartoons are used similar to forward-chaining rules. Contrary to the “declarative” forward-chainers based on Prolog [30, 78], cartoons *are* completely declarative. The Prolog-based versions are more flexible than the current implementation of the Cartoonist engine because a rule’s condition can take advantage of the Prolog reasoning engine whereas a cartoon is restricted to a simple pattern. However, there is no reason why the patterns could not in principle be Prolog predicates. Even the description of the future state could use Prolog predicates, whereas the actions of the Prolog-based versions are restricted to asserting and retracting facts.

### 6.1.2.2 Constraint Programming

In a cartoon, each state is described by a constraint. The constraints are sequentially ordered, and thus a cartoon is a special case of a temporal constraint. The variables in the constraints are restricted by the relations in a constraint, for instance, a certain square must be empty. The variables are also restricted by the sequential ordering of the constraints; for instance, switching two state constraints in a cartoon may cause an object to move into a different direction.

The Cartoonist engine could possibly be implemented with constraint programming by formulating the Cartoonist description of a simulation as a temporal constraint problem that has a sequence of actions as a solution. A potential problem of this approach is that a simulation may have no clear end, and therefore the number of actions in the solution would be infinite.

### 6.1.2.3 Planning

The Cartoonist engine generates a sequence of actions whose initial and last states only are displayed. To the user, these actions appear to be executed in parallel. Although computing this action sequence might look similar to planning, it is not. The order of the action is specified by the cartoons that apply, and if more than one action has as good a support

as the best action, then one of the actions is chosen randomly even if it may make a good action in the future impossible.

The reason is that the system has a look-ahead of only one state. If the definition of a cartoon would not require that the second-to-last constraint describes the current state, then the Cartoonist engine would become more similar to a planner. The constraints ranging further into the future would then become more like a goal. However, it could seriously reduce the efficiency of the Cartoonist engine.

### 6.1.3 Object-Oriented Programming

The rules that specify the character's actions are similar to methods in object-oriented programming. In the Agentsheets-Cartoonist implementation, most rules translate into one method. There are exceptions, because rules are not restricted to one call, for instance, castling in chess would lead to a call to the king and one to the rook.

In visual programming systems, the objects in the pictures manipulated by the rules consist generally only of one or more pictures and possibly some slots containing values. In Cartoonist, the objects contain also actions defining some of the objects' semantics. For instance, a ball cannot move through walls nor be beamed to a remote location, and walls cannot move.

The castling example suggests sending a method to a dynamically formed group of objects. Should the message be sent to the rook or to the king, or should a further object represent a rook/king pair? In Cartoonist the problem is solved using cartoons that are not directly associated with a specific object but with a dynamically formed group.

Although cartoons generally are used in these situations, rules can be used too, as the castling example showed. For instance, ChemTrains [5] uses this kind of global rule, whereas the Agentsheets [63] implementation associates each rule with a specific object class, thus having the problem described above. However, the latter approach tends to be more efficient and somewhat more flexible.

Cartoonist provides visual objects whose semantics are partially defined by the rules, yet they still provide the global cartoons to describe behavior that cannot be easily attributed to a specific object.

### 6.1.4 Visual Programming Languages

Cartoonist is independent of a specific visual representation of the cartoons and rules. In fact, the Cartoonist engine is purely textual. However, due to the state-based representation of the cartoons, cartoons are well suited as mechanisms for visual programming.

### 6.1.5 Programming by Example

In Agentsheets-Cartoonist, the cartoons are created similar to most other visual rule-based programming systems by selecting a part of the simulation display as the first picture in the cartoon. Cartoons generally look less like an example than do rewrite rules, especially if high-level relations are used in the constraints.

Similar to several programming-by-demonstration systems, Cartoonist uses comic strip-like snapshot sequences to describe the program. Cartoonist uses these snapshots to guide the choice of actions dynamically depending, on the current situation; the other systems infer a program from the snapshots at definition time.

### 6.1.6 Programming with Temporal Information

Few rule-based systems allow the user to access past states in the condition. As Chapter 5 shows, it is not difficult to build such a system, and its power to program simulations might be considerable. Instead of constraining the sequence of states with cartoons, Mitchell uses temporal logic formulas to constrain the sequence of allowed rule applications [54]. His work is theoretical and its value in a system is unclear.

### 6.1.7 Behavior-Based Programming

In behavior-based programming, the user describes the simulation with cartoons in terms of behaviors rather than actions. These behaviors can then be combined similarly to the way it is done in the subsumption architecture. As the avoidance example illustrates (see Section 3.4.5), combining the cartoons dynamically depending, on the current and the past situations, is a powerful way of iterative program development.

## 6.2 Results

The introduction of this dissertation listed the following four problems:

- composing complex behavior from separate descriptions,
- dealing with temporal concepts,
- specifying a wide range of parallel behaviors,
- describing behavior by specifying what should *not* happen.

The goal of the dissertation was to find a rule based-like system that solves these four problems while retaining the advantages of rule-based systems. This section will discuss the main contributions of this dissertation, namely the solutions to these problems

### 6.2.1 Retaining the Advantages of Rule-Based Systems

The advantages of rule-based systems that are important for visual end-user programming are the simplicity of the rules and their ability to describe state changes with before- and after-pictures. These systems also provide a uniformity because only rules are used to describe a simulation.

Cartoonist is not that uniform because it uses rules and cartoons. However, rules are rarely used and tend to be much simpler than the rules in rules-only systems. Cartoons are similar to rules but are used differently. User testing must show whether end users can deal with cartoons as well as with rules. Since cartoons are more declarative than rules, end users may even have fewer problems with cartoons than with rules.

### 6.2.2 Composing Behaviors

The subsumption-like architecture of Cartoonist allows the user to incrementally refine the behavior of the simulation objects. The interaction and independence between cartoons is quite different from the one between rules, leading to rather different descriptions of the same behavior.

The description of the simulation can be developed in layers, where each layer gets the simulation a bit closer to the desired behavior. Rules require more of an all-or-nothing approach, in which a description of the simulation must be correct from the beginning and is difficult to refine.

### 6.2.3 Temporal Concepts

Temporal concepts, such as moving in the same direction, can be easily described with a cartoon, but rules need additional representational features to describe the same behavior. Although these temporal concepts are relatively simple, they are of great value to describe simulations, because complex interactions between objects can be described quite easily.

### 6.2.4 Parallel Behavior

Cartoonist can describe a wide range of parallel behavior. The declarative nature of cartoons makes it possible to describe parallel actions without having to refer to the concept of

parallelism explicitly. However, finding the right description is not always that easy, although the final descriptions tend to be simple.

### 6.2.5 Negative Programming

Since cartoons describe the future state and not the action to reach this state, the description is purely declarative. The cartoon recognizes the next state; it does not generate it. Any kind of predicate or relation can be used to describe the future state. Thus, negation can be used to describe those states that should not be reached with the next action.

## 6.3 Future Work

Future research on the Cartoonist framework consists of building an efficient and user-friendly development environment that can be used to gain more experience with cartoon-based programming and to do extensive user testing. The experimental data will help guide the further improvement of Cartoonist.

The new implementation should not necessarily be restricted to grids, but may be combined with topological structures as in ChemTrains. Extensions to the definition of a cartoon should also be considered. Possibilities are:

- The states described by a cartoon are not required to follow each other immediately, that is, use the sooner-or-later temporal operator. For instance, describing that spring, summer, fall, and winter follow each other in this order is currently not possible with just one cartoon.
- Do not restrict the second-to-last constraint to describing the current state. This will allow the cartoon to describe more than just one future state. The system would become more of a planning character. However, this may be problematic due to its potential computational complexity.
- The whole state may be negated. This is not the same as negative programming in which only conjuncts may be negated but not the whole constraint. Currently, this can be accomplished with a non-primitive predicate representing the logical formula; however, this is not flexible.
- The definition of non-primitive predicates and relations should be supported. Currently only negation is available.

# Bibliography

- [1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *AAAI-87*, pages 268–272, 1987.
- [2] Rudolf Arnheim. Learning by looking and thinking. *Educational Horizons*, 71(2):94–98, 1993.
- [3] E. Bahr, F. Barachini, J. Doppelbauer, H. Gräbner, F. Kaspavec, T. Mandl, and H. Mittelberger. A parallel production system architecture. *Journal of Parallel and Distributed Computing*, 13:456–462, 1991.
- [4] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 2. Kaufmann, Los Altos, CA, 1982.
- [5] Brigham Bell. Using programming walkthroughs to design a visual language. Technical Report CU-CS-581-92, Department of Computer Science, University of Colorado at Boulder, February 1992.
- [6] Brigham Bell and Clayton Lewis. Chemtrains: A language for creating behaving pictures. In *Proc. 1993 IEEE Symposium Visual Languages*, pages 188–195, Bergen, Norway, 1993.
- [7] A. Biermann. Automatic programming. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 18–35. John Wiley & Sons, New York, 1990.
- [8] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.
- [9] Ronald J. Brachman and Hector J. Levesque. *Readings in Knowledge Representation*. Morgan Kaufmann, San Mateo, CA, 1985.
- [10] Rodney A. Brooks. Intelligence without reason. Technical Report 1293, Massachusetts Institute of Technology, A.I. Laboratory, 1991.



- [11] F. P. Brooks Jr. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.
- [12] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming in Expert Systems: An Introduction to Rule-Based Programming*. Addison-Wesley, Reading, MA, 1985.
- [13] Christoph Brzoska. Temporal logic programming and its relation to constraint logic programming. In *International Symposium on Logic Programming*, San Diego, 1991.
- [14] Margaret M. Burnett and Allen L. Ambler. Declarative visual languages. *Journal of Visual Languages & Computing*, 5:1–3, 1994.
- [15] Margaret M. Burnett and Marla J. Baker. A classification system for visual programming languages. Technical Report 93-60-14, Department of Computer Science, Oregon State University, Corvallis, OR 97331, 1993.
- [16] S.-K. Chang, T. Ichikawa, and P. Ligomenides. *Visual Languages*. Plenum Press, New York, 1986.
- [17] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.
- [18] William J. Clancey and Conrad Block. Representing control knowledge as abstract tasks and metarules. In L. Bolc and M. J. Coombs, editors, *Expert System Applications*, pages 1–77. Springer-Verlag, New York, 1988.
- [19] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.
- [20] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [21] Donald Crafton. *Emile Cohl, Caricature, and Film*. Princeton University Press, Princeton, NJ, 1990.
- [22] Nancy Cunniff and Robert P. Taylor. Graphical vs. textual representation: An empirical study of novices' program comprehension. In G. M. Olson and S. Sheppard and E. Soloway, editors, *Empirical Studies of Programmers*, pages 114–131. Washington, D.C., December 1987.

- [23] Allen Cyphert. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, 1993.
- [24] Allen Cyphert and David Canfield Smith. KidSim: End user programming of simulations. In *CHI'95*. ACM, 1995.
- [25] Randall Davies. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15:179–222, 1980.
- [26] Charles Forgy. Rete: A fast algorithm for many pattern / many object match problem. *Artificial Intelligence*, 19:17–37, 1982.
- [27] Martin R. Frank and James D. Foley. A pure reasoning engine for programming by example. Technical Report git-gvu-94-11, Georgia Institute of Technology, April 1994.
- [28] George 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.
- [29] George W. Furnas. Reasoning with diagrams only. In *AAAI Symposium on Reasoning with Diagrammatic Representations*. Stanford, CA, March 1992.
- [30] Mauro Gaspari. Extending Prolog with data driven rules. Technical Report UBLCS-94-3, Laboratory for Computer Science, University of Bologna, Italy, 1994.
- [31] Michael R. Genesereth. An overview of MRS for AI experts. Technical Report HPP-82-27, Department of Computer Science, Stanford University, January 1984.
- [32] Ephraim P. Glinert. *Visual Programming Environments*. IEEE Computer Society Press, Los Alamitos, CA, 1990.
- [33] Joseph Y. Halpern. A guide to the modal logics of knowledge and belief: Preliminary draft. In *IJCAI*, pages 480–490, 1985.
- [34] Wilson Harvey, Dirk Kalp, Milind Tambe, David McKeown, and Allen Newell. The effectiveness of task-level parallelism for production systems. *Journal of Parallel and Distributed Computing*, 13:395–411, 1991.
- [35] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1983.

- [36] Andreas Huber. *Die Entwicklung praxisorientierter wissensbasierter Systeme – Diskussion methodischer Fragen am Beispiel des Banker’s Assistant*. PhD thesis, Department of Computer Science, University of Zurich, February 1993.
- [37] Roland Hübscher, Clayton Lewis, and Alex Repenning, editors. *Child’s Play: A Hands-On Workshop on End-User Programming*, Department of Computer Science, University of Colorado at Boulder, 1995.
- [38] Toru Ishida. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1), 1991.
- [39] Wolfram Knorr. Als die Bilder sprechen lernten. *Die Weltwoche*, pages 53–57, April 13 1995.
- [40] Kenneth R. Koedinger. Emergent properties and structural constraints: Advantages of diagrammatic representations for reasoning and learning. In *AAAI Spring Symposia on Reasoning with Diagrammatic Representations*. Stanford University, 1992.
- [41] Chin-Ming Kuo. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13:424–441, 1991.
- [42] Steve Kuo and Dan Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal of Parallel and Distributed Computing*, 13:383–394, 1991.
- [43] David Joshua Kurlander. Graphical editing by example. Technical Report CUCS-023-93, Columbia University, July 1993.
- [44] David Joshua Kurlander and Steven Feiner. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics*, 1993.
- [45] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.
- [46] F. Lakin. Spatial parsing for visual languages. In S.-K. Chang and T. Ichikawa and P. Ligomenides, editors, *Visual Languages*, pages 35–85. Plenum Press, New York, 1986.
- [47] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.
- [48] Clayton Lewis. Pictorial rewrite rules and kids’ science. In R. Hübscher and C. Lewis and A. Repenning, editors, *Child’s Play: A Hands-On Workshop on End-User Programming*. Boulder, 1995.

- [49] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
- [50] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Kaleidoscope: A constraint imperative programming language. Technical Report 93-09-04, Department of Computer Science and Engineering, University of Washington, September 1993.
- [51] David McIntyre and Ephraim P. Glinert. Visual tools for generating iconic programming environments. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 162–168, 1992.
- [52] Jeffrey D. McWhirter. *Characterization, Specification and Generation of Visual Language Applications*. PhD thesis, Department of Computer Science, University of Colorado at Boulder, 1994.
- [53] Daniel P. Miranker. Treat: A better match algorithm for ai production systems. In *AAAI-87*, pages 42–47, Seattle, WA, 1987.
- [54] William P. R. Mitchell. Temporal term rewriting. Technical Report UMCS-88-4-1, Department of Computer Science, University of Manchester, 1988.
- [55] Francesmary Modugno and Brad Myers. Visual representations as feedback in a programmable visual shell. Technical Report CMU-CS-93-133, Carnegie Mellon University, School of Computer Science, 1993.
- [56] Brad Myers. Taxonomies of visual programming and program visualisation. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.
- [57] Bonnie A. Nardi. *A Small Matter of Programming : Perspectives on End User Computing*. MIT Press, Cambridge, MA, 1993.
- [58] Allen Newell. The knowledge level. *Artificial Intelligence*, 18(1):82–127, 1982.
- [59] Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.
- [60] Mark Perlin. Constraint-based specification of production rules. In *IEEE International Workshop on Tools for Artificial Intelligence*, pages 332–338, Fairfax, VA, 1989. IEEE Computer Society Press.

- [61] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.
- [62] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer Verlag, New York, 1990.
- [63] Alex Repenning. Agentsheets: A tool for building domain-oriented dynamic, visual environments. Technical Report CU-CS-693-93, Department of Computer Science, University of Colorado at Boulder, December 1993.
- [64] Alex Repenning. Bending icons: Syntactic and semantic transformation of icons. In *IEEE Workshop on Visual Languages*, St. Louis, MO, 1994. IEEE.
- [65] Alex Repenning. Bending the rules: Steps toward semantically enriched graphical rewrite rules. In *IEEE Workshop on Visual Languages*, 1995.
- [66] Alex Repenning and Tammara Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *Computer*, 28:17–25, 1995.
- [67] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
- [68] James G. Schmolze. Guaranteeing serializable result in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13:348–365, 1991.
- [69] Andy Schürr. PROGRES, a visual languages and environment for PROgramming with Graph REwriting Systems. Technical Report AIB 94-11, Lehrstuhl für Informatik, Technische Hochschule Aachen, 1994.
- [70] N.C. Shu. *Visual Programming*. Van Nostrand Reinhold, 1988.
- [71] Brian Cantwell Smith. Prologue to “reflection and semantics in a procedural language”. In R. J. Brachman and H. J. Levesque, editors, *Readings in Knowledge Representation*, pages 31–40, San Mateo, CA, 1985. Morgan Kaufmann.
- [72] David Canfield Smith, Allen Cypher, and James Spohrer. KidSim: Programming agents without a programming language. *Communications of the ACM*, 37(7), 1994.
- [73] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, 1994.

- [74] Salvatore J. Stolfo, Ouri Wolfson, Philip K. Chan, Hasanat M. Dewan, Leland Woodbury, Jason S. Glazier, and David A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13:366–382, 1991.
- [75] Milind Tambe, Allen Newell, and Paul S. Rosenbloom. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5:299–348, 1990.
- [76] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint programming. *Artificial Intelligence*, 58:113–159, 1992.
- [77] S. Vere. Planning. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 748–758. John Wiley & Sons, New York, 1990.
- [78] Steve Wallis and Scott Moss. Efficient forward chaining for declarative rules in a multi-agent modelling language. Technical Report CPM Report 004, Centre for Policy Modelling, Manchester Metropolitan University, UK, 1994.
- [79] *Webster's New Universal Unabridged Dictionary*. Dorset & Barber, 2nd edition, 1983.



# Appendix A

## A More General Treatment of Temporal Constraints

This appendix can be skipped without missing information about the cartoons and their use. It presents a more general treatment of temporal constraints than in the main part of this dissertation. The concept of a cartoon was developed by reflecting upon temporal constraints, the subsumption architecture, and hierarchy of cartoons.

This appendix introduces formally temporal constraints, a generalization of cartoons. This more general treatment of temporal constraints allows the discussion of possible extensions of cartoons. In addition, the subsumption architecture preference scheme is discussed. It is implemented in the cartoon version with voters. The voting concept is presented in a more detailed form than in Chapter 2 which provides some suggestions for possible extension of the voting process.

### A.1 Temporal Constraints

This section defines temporal constraints. One of the many special cases of temporal constraints is the cartoon as treated in the body of the thesis. The use of temporal constraints as a basis for cartoons is useful because it allows comparison of different versions and extensions of cartoons as they are defined. Although the temporal constraints as defined here are much more general than the cartoons, they are not as powerful as many of the related first-order temporal logics [13, 33].

The end user will never have to deal with the definition of a temporal constraint. All that is shown to the end user of a Cartoonist system is a user-friendly description of a cartoon—and this will certainly not be the formal definition.



### A.1.1 Formal Definition

A simulation is an ordered sequence of states. The *visible* state is shown in the simulation display and the *invisible* state is not shown. This makes it possible to control a wide range of parallel behavior in a flexible way. A temporal constraint describes a sequence of states  $s_1, s_2, \dots, s_k$ , where  $t(s_i) \leq t(s_{i+1})$  for all  $i$ ,  $1 \leq i < k$  and  $t(s)$  is the time in state  $s$ . Time in an invisible state is always the same as in the last previous visible state. Any visible state is one time step later than the previous visible state. For instance, in the sequence  $s_1^v, s_2^v, s_3^u, s_4^u, s_5^v$  with the visible states  $s_1^v, s_2^v, s_5^v$  and the invisible states  $s_3^u, s_4^u$  the following holds:

$$t(s_1^v) + 1 = t(s_2^v) = t(s_3^u) = t(s_4^u) = t(s_5^v) - 1$$

**CONSTRAINT:** A constraint  $C$  is a set of relations. The constraint  $C$  is satisfied in a state  $s$  if  $s$  is consistent with  $C$ .  $C(s, \sigma)$  means that state  $s$  is consistent with constraint  $C$  given the variable substitution  $\sigma$ .

**TEMPORAL CONSTRAINT:** A temporal constraint is of the form

$$C_1 \rightarrow_{o_1} C_2 \rightarrow_{o_2} \dots \rightarrow_{o_{n-1}} C_n,$$

where each  $C_i$ ,  $1 \leq i \leq n$  is a constraint and the  $\rightarrow_{o_i}$  are temporal operators.

The temporal operators  $\rightarrow_{o_i}$  have an index because there exists a great variety of different operators. A temporal operator specifies the temporal relation between the states and whether the states are visible or invisible.

A temporal operator is defined using a regular language. A regular language is closed under the operations of union ( $\cup$ ), concatenation ( $\circ$ ), and Kleene star ( $*$ ) [49]. Assume that the operator should accept a state sequence consisting of a visible state, followed by and invisible and then a visible one. Let  $v$  stand for visible and  $u$  for invisible. (The choice of  $u$  for invisible is based on “invisible” and trying to avoid confusion due to the frequent use of  $i$  as an index.) Then, the visibility of the states in an accepted state sequence can be abbreviated with  $vuv$ . The language defined by the regular expression  $vuv$  is used to define the temporal operator as follows.

**TEMPORAL OPERATOR:** A temporal operator ( $\rightarrow$ ) applied to two constraints  $C_1 \rightarrow C_2$  defines what state sequences  $s_1, \dots, s_k$  are accepted with respect to the states’ visibility

assuming that  $s_1$  and  $s_k$  are consistent with  $C_1$  and  $C_2$ , respectively. The acceptable state sequences are defined by a regular expression over the alphabet  $\{v, i\}$ .

The language associated with a temporal constraint  $\rightarrow$  is  $\mathcal{L}(\rightarrow)$ . Let  $R(s)$  be defined such that it returns the letter  $v$  if  $s$  is visible and the letter  $u$  if  $s$  is invisible. A state sequence  $s_1, s_2, \dots, s_k$  is described by a temporal constraint  $C_1 \rightarrow C_2$  if

- (1)  $C_1$  is consistent with  $s_1$ ,
- (2)  $C_2$  is consistent with  $s_k$ ,
- (3) for all intermediate states  $s_i$ ,  $1 < i < k$ ,  $C_2$  is not consistent with  $s_i$ ,
- (4) the word  $R(s_1)R(s_2) \cdots R(s_k)$  is in  $\mathcal{L}(\rightarrow)$ .

These four conditions can be formalized as follows:

$$\exists \sigma. C_1(s_1, \sigma) \wedge C_2(s_k, \sigma) \wedge \forall i, 1 < i < k. \neg C_2(s_i, \sigma) \wedge R(s_1)R(s_2) \cdots R(s_k) \in \mathcal{L}(\rightarrow)$$

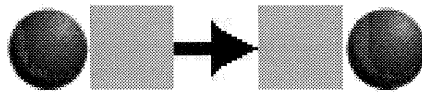
**Example** Let  $\mathcal{L}(\rightarrow)$  be the language described by the regular expression  $v \circ u^* \circ v$ , or  $vu^*v$  for short. Then, the state sequence  $s_1, \dots, s_k$  is described by  $C_1 \rightarrow C_2$  if  $C_1$  and  $C_2$  describe  $s_1$  and  $s_k$ , respectively.  $C_2$  does not describe any of  $s_2, \dots, s_{k-1}$ , and  $s_1, s_k$  are the only two visible states in the sequence. This means,  $R(s_1)R(s_2) \cdots R(s_k)$  is of the form  $vu \cdots uv$ .

A temporal constraint is not restricted to two constraints only. The given definition can be easily generalized to a temporal constraint of the form  $C_1 \rightarrow_{o_1} C_2 \rightarrow_{o_2} \cdots \rightarrow_{o_{n-1}} C_n$ . The temporal constraint describes the state sequence  $s_{k_1}, s_{k_1+1}, \dots, s_{k_2}, \dots, s_{k_n}$  if each temporal constraint  $C_i \rightarrow_{o_i} C_{i+1}$  in the original temporal constraint describes the sequence of states  $s_{k_i}, \dots, s_{k_{i+1}}$ . It is necessary that each time the same variable substitution is used, that is,  $\exists \sigma \forall i, 1 \leq i < k. C_i(s_{k_i}, \sigma)$ .

### A.1.2 Temporal Operators and Invisible States

Invisible states are like visible states except they are not shown to the end user in the simulation display. They exist only internally and the user has no knowledge of them. This makes it possible, together with the correct temporal operators, to describe a wide range of parallel actions, as is shown in Chapter 4. Although the user has the impression that several actions are executed in parallel, internally everything is sequential, which simplifies the Cartoonist engine considerably.

The first operator is defined by the regular expression  $vv$  and is denoted  $\mapsto$ . This means the state sequence consists of two visible states  $s_1, s_2$  such that  $\exists \sigma. C_1(s_1, \sigma) \wedge C_2(s_2, \sigma)$ . In other words, two temporally contiguous visible states are consistent with  $C_1$  and  $C_2$ .



**Figure A.1:** Visual version of the rule in Example 1. A gray square means the square must be empty.

The second operator, denoted  $\Rightarrow$ , is defined by  $vu^*$ , the *sooner-or-later* operator. It is the same as the previous operator  $\mapsto$  except that the two states are not required to follow each other immediately.

The third and final operator, denoted  $\searrow$ , is used for describing parallel actions and is defined by  $vu^*(v \cup u)$ .  $C_1$  always describes a visible state whereas  $C_2$  may describe a visible or an invisible state. However, no visible state is allowed between the first and the last state. Since invisible states are not displayed, this operator makes it possible to describe exactly which actions should be executed in parallel.

Assuming  $C_1 \rightarrow C_2$  describes a sequence of at most  $k$  states, then  $\prod_{i=1}^k 2^i \approx 2^{k^2/2}$  different temporal operators are possible, because  $2^i$  different strings of length  $i$  exist over the alphabet  $\{v, u\}$ . Most of them are not interesting, but the previous three operators are useful for describing simulations.

### A.1.3 Representation of Temporal Concepts

Temporal concepts are important in describing a simulation because changes over time, especially movements and interactions between characters, are so common. A few examples will show how the different temporal operators can be used to describe temporal concepts.

The form of the temporal constraints that are allowed is slightly restricted. A temporal constraint  $C_1 \rightarrow_{o_1} C_2 \rightarrow_{o_2} \dots \rightarrow_{o_{n-1}} C_n \rightarrow_{o_n} C_{n+1}$  describes a state sequence only if  $C_n$  describes a part of the *current* state. This implies that only one state in the future is described and  $n - 1$  past states are referred to.

Spatial primitives use a grid, in which each character is located in one square of the grid. Small italicized letters, possibly indexed, are variables. Let  $L(x, y)$  stand for “ $x$  is on the left of  $y$ ” and  $R(x, y)$ ,  $U(x, y)$ ,  $D(x, y)$  similarly denote right, up, and down, respectively.  $T(a, r, x, y)$  means that character  $a$  of type  $r$  is at location  $(x, y)$ . Type  $r$  is, for instance, ball, wall, ice, water, and so on, and  $a$  is a unique identifier of the character.

**Example 1** The problem is to keep a ball moving to the right. The solution is trivial and similar to a rule-based solution. The visual version is shown in Figure A.1 and the textual one is shown in Figure A.2.

$$\{T(b, \text{ball}, x_1, y)\} \mapsto \{T(b, \text{ball}, x_2, y), L(x_1, x_2)\}$$

Figure A.2: Textual solution to Example 1.

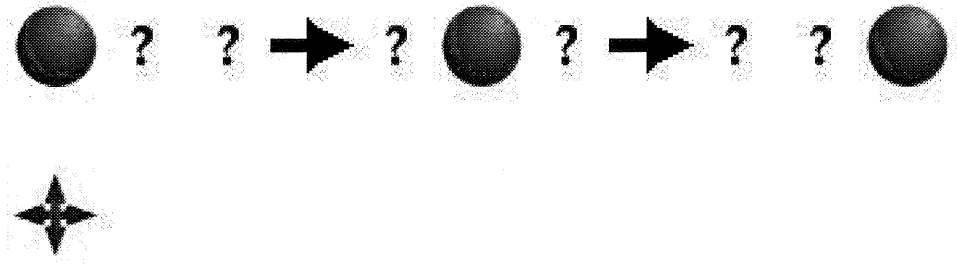


Figure A.3: Visual representation of the four temporal constraints for Example 2.

$$\begin{aligned}
 (1) \quad \{T(b, \text{ball}, x_1, y)\} &\mapsto \{T(b, \text{ball}, x_2, y), L(x_1, x_2)\} \\
 &\mapsto \{T(b, \text{ball}, x_3, y), L(x_2, x_3)\} \\
 (2) \quad \{T(b, \text{ball}, x_1, y)\} &\mapsto \{T(b, \text{ball}, x_2, y), R(x_1, x_2)\} \\
 &\mapsto \{T(b, \text{ball}, x_3, y), R(x_2, x_3)\} \\
 (3) \quad \{T(b, \text{ball}, x, y_1)\} &\mapsto \{T(b, \text{ball}, x, y_2), U(y_1, y_2)\} \\
 &\mapsto \{T(b, \text{ball}, x, y_3), U(y_2, y_3)\} \\
 (4) \quad \{T(b, \text{ball}, x, y_1)\} &\mapsto \{T(b, \text{ball}, x, y_2), D(y_1, y_2)\} \\
 &\mapsto \{T(b, \text{ball}, x, y_3), D(y_2, y_3)\}
 \end{aligned}$$

Figure A.4: Textual solution for Example 2.

The textual solution is also provided to make explicit what the visual representation means. The meaning is defined on the textual representation because the algorithm in Cartoonist reasons on the textual representation only.

**Example 2** What if the ball should keep going in the same direction, independent of the direction? In this case, rules are problematic and require the addition of an explicit feature representing the current direction and whether the ball is moving at all. Temporal constraints solve this problem easily. There is one temporal constraint for each of the four directions, as Figure A.4 shows. One of the four temporal constraints plus information that the temporal constraint applies in all four directions is sufficient to generate the other three temporal constraints because the three additional temporal constraints are different only with respect to symmetries. See Figure A.3 for the visual version of the four temporal constraints.

These temporal constraints represent the concept of impetus. If the ball is initially not moving, then it will not start moving. This is not true for the temporal constraint in

$$\begin{aligned} \{T(b, \text{ball}, x_1, y)\} &\Rightarrow \{T(b, \text{ball}, x_2, y), L(x_1, x_2)\} \\ &\Rightarrow \{T(b, \text{ball}, x_3, y), L(x_2, x_3)\} \end{aligned}$$

**Figure A.5:** A solution of Example 3.

$$\begin{aligned} \{T(b, \text{ball}, x_1, y)\} &\Rightarrow \{T(b, \text{ball}, x_2, y), L(x_1, x_2)\} \\ &\searrow \{T(b, \text{ball}, x_3, y), L(x_2, x_3)\} \end{aligned}$$

**Figure A.6:** A solution for Example 4.

Example 1. In both examples, the operator  $\mapsto$  was used, which connects two visible states that follow each other immediately in time.

**Example 3** Examples 1 and 2 can be combined. A ball, once moving in a direction, keeps moving in this direction even if it does not move at each time step.

The solution in Figure A.5 is equivalent to the first one, except that the temporal operator  $\Rightarrow$  is used instead of  $\mapsto$ . The operator  $\Rightarrow$  connects two visible states, however, the states are not required to follow each other immediately in time.

The problem with all these solutions is that they cannot deal with more than one ball in parallel because no invisible states are possible. The next example takes care of this problem.

**Example 4** This problem is the same as in Example 2, except that it works for more than one ball.

The solution in Example 2 works with only one ball because the ball has to move in every time step to keep moving. Solutions from either Example 2 or Example 3 can be used and the second temporal operator is replaced with  $\searrow$ . So, for instance, the temporal constraint in Figure A.6 is a solution.

These examples show clearly that a variety of temporal operators are interesting. The next section will show that these temporal operators can be hidden from the end user by using cartoons instead of the more general temporal constraints.

## A.2 Cartoons as Temporal Constraints

Temporal constraints are a powerful tool to describe a great variety of state sequences. However, they are not easy to understand, and end users cannot be expected to use a set of

different temporal operators. Therefore, a subset of temporal constraints is chosen that is powerful yet can be presented to the end user in a simple form. The concept of a cartoon defines this subset.

As said earlier, a cartoon is a special case of a temporal constraint. The previously introduced three temporal operators already suggest what might be the best definition of a cartoon. What led to the definition below is a superset of the problems presented in Chapter 3.

The goal is to find a simple definition that enables a user to solve many interesting problems. It has to be simple for the end user. That is, the fewer the kinds of temporal operators and the simpler their semantics are, the better. The following definition satisfies these criteria.

CARTOON: A *cartoon* is a temporal constraint of the form

$$C_1 \mapsto C_2 \mapsto \cdots \mapsto C_k \searrow C_{k+1},$$

where  $k \geq 1$ , and  $C_k$  describes the current state.

From now on, ‘ $\rightarrow$ ’ will be used for all temporal operators in a cartoon; that is, the above cartoon is normally written as  $C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_k \rightarrow C_{k+1}$ .

As described earlier, the temporal operator  $\mapsto$  is defined by the regular expression  $vv$  and  $\searrow$  by  $vu^*(v \cup u)$ . A cartoon therefore describes a sequence of states  $s_1, s_2, \dots, s_k, \dots, s_h$ , where all  $s_i$ ,  $1 \leq i \leq k$ , are visible, all states  $s_j$ ,  $k < j < h$  are invisible, and  $s_h$  is either visible or invisible. By definition,  $s_k$  is the current state, and thus  $s_1, \dots, s_{k-1}$  are past states following each other immediately in time.

The constraints  $C_1, \dots, C_k$  can be viewed as the condition referring to the current state and an arbitrarily long sequence of past states.  $C_{k+1}$  describes the immediately following state in the future. Since the operator between  $C_k$  and  $C_{k+1}$  is described by the regular expression  $vu^*(v \cup u)$ , the constraint  $C_{k+1}$  describes parts of a set of parallel actions. Depending on the present and past states, cartoons describe parts of a mapping from the current to the next state consisting of a set of parallel actions that are executed sequentially. A detailed discussion of the different kinds of parallelism, including the “sequential parallelism” used by Cartoonist, can be found in Chapter 4.

### A.3 Subsumption Architectures

Brooks has proposed a computational model, called the *subsumption architecture*, that is well suited for situated action [10]. The subsumption architecture is based on the idea that in intelligent, situated, and embodied agents such as robots, reasoning and explicit representations of the world should be replaced by descriptions of behavior. This is quite different from the more conventional view, which explains behavior as being caused by the interpretation of an internal and external representation by a reasoning engine.

One of the key features of the subsumption architecture is a network of behaviors (or computations) that can inhibit each other. Possible behaviors of a robot are, ordered from low to high level: avoid objects, follow walls, explore the environment, identify objects, and so on. Each of these behaviors has direct access to the sensors and the actuators of the robot. However, higher-level behaviors have access to the internal states of the lower-level behaviors and can suppress or modify their output. There is no hierarchy in the sense of goal/subgoal, but the computations can be combined if they are compatible.

As long as the high-level behavior “identify object” does not become active or is consistent with the lower-level behavior “follow wall,” the robot may follow a wall. The two behaviors may even be combined if both behaviors are triggered by the sensors.

In Cartoonist, behavior is combined similar to the robot’s subsumption architecture. It accounts for the flexible refinement of already existing behavior, as in the original subsumption architecture, and therefore I have chosen to use the same name for Cartoonist’s architecture.

### A.4 Preference Scheme

The preference scheme consisting of a dynamic and static preference is defined for cartoons. The discussion of the preference is again kept at a general level to provide insights for possible extensions of the Cartoonist framework in the future.

Cartoons are organized by a preference relation  $\succ$ . The static part of this relation is defined by the user, the creator of the cartoons; the dynamic part is dependent on how the cartoon describes a state sequence.

After the necessary characteristics of the preference relation are motivated, the relation is defined for the general case of a temporal constraint. Then, it is specialized for cartoons, which leads to a simple definition of the preference relation.

### A.4.1 Necessary Characteristics

A *cartoon instance* is a cartoon that is associated with a state sequence, which is described by an initial sequence of constraints of the cartoon. The definition for the preference relation  $\succ$  must have following characteristics ordered with respect to their priorities.

1. The user overrides any dynamic preference scheme.
2. The user should state preferences between pairs of cartoons and should not be required to completely order all the cartoons.
3. A cartoon instance is more important the longer its past is, that is, the more past states it describes.
4. A cartoon instance is more important the farther into the future it looks, that is, the more future states it describes.

The first item of the list requires that the system must not change the user's preference scheme, whatever it uses internally to break ties. This is accomplished by the definition of  $\succ$  by considering only the dynamic preference if the static preference is not decisive.

Item (2) suggests a partial preference relation between cartoon instances and not an ordered, linear list, as in ChemTrains or ScienceShows. Most of the time only a few cartoons interact, and this should not require the user to order all of the cartoons in one list. Using a partial order, groups of cartoons can be ordered flexibly and locally. A total order is, of course, still possible; however, it is not required. By default, cartoons are not ordered. Preferences can be added on demand. This is rarely necessary, as the examples in Chapter 1 show.

The third and fourth requirements are based on the analysis of a more complicated version of cartoons than defined here applied to the Towers of Hanoi example. The idea is that a cartoon can be viewed as a procedure that is invoked in a state  $s$ , if the first constraint of the cartoon is consistent with  $s$ . The cartoon “returns” if its last constraint describes a state. This suggests the third and fourth requirement which will be discussed in more detail after the preference relation has been defined.

### A.4.2 Formal Definition

**STATIC PREFERENCE RELATION  $\succ_s$ :** The static preference is defined by the user by explicitly stating with  $\alpha \succ_s \beta$  that cartoon  $\alpha$  is statically preferred over cartoon  $\beta$ . Let  $\sigma$  and



$\mu$  be variable substitutions. For the instances  $\alpha^\sigma$  and  $\beta^\mu$  of these cartoons this means that

$$\alpha^\sigma \succ_s \beta^\mu \equiv \alpha \succ_s \beta$$

Two cartoon instances are statically indifferent,  $\alpha^\sigma \sim_s \beta^\mu$ , if the user has not added a preference relation between the two cartoons  $\alpha$  and  $\beta$ .

$$\alpha^\sigma \sim_s \beta^\mu \equiv \neg(\alpha \succ_s \beta \vee \beta \succ_s \alpha)$$

The definition of the dynamic preference between two cartoons requires a few preliminary notational definitions. Let  $s_a^\alpha$  and  $s_z^\alpha$  be the first and last states, respectively, described by the cartoon instance  $\alpha^\sigma$ , where  $\alpha$  is the constraint  $C_1 \rightarrow \dots \rightarrow C_i \rightarrow \dots \rightarrow C_k$ . (In  $s_a^\alpha$  the  $\alpha$  returns to the cartoon and the index  $a$  refers to the first state whereas in  $s_z^\alpha$   $z$  refers to the last state. Further assume that  $C_i$  describes  $s_z^\alpha$  and the current state is  $s$ . By definition,  $C_1$  describes  $s_a^\alpha$ . The time of a state  $s$  is denoted by  $t(s)$ , and it follows from the definition of  $s$ ,  $s_a^\alpha$ , and  $s_z^\alpha$  that  $t(s_a^\alpha) \leq t(s_z^\alpha) \leq t(s)$ .

Function  $\Delta(\gamma)$  computes the minimal number of visual states that are described by  $\gamma$ , which is a suffix of a cartoon. A suffix of a cartoon  $C_1 \rightarrow \dots \rightarrow C_k$  is a cartoon  $C_i \rightarrow \dots \rightarrow C_k$  where  $j \geq 1$ . Assume that the current state is consistent with the first constraint in  $\gamma$  and  $t$  is the current time. Then,  $t + \Delta(\gamma) - 1$  is the earliest point in time at which the cartoon describes a state sequence leading to a vote.

$\Delta(\gamma)$  returns the length of the minimal number of visible states in the shortest possible state sequence described by the suffix of a temporal constraint  $\gamma$ , assuming that the last state in the sequence must be visible. The latter condition is required because  $\gamma$  is a cartoon suffix; that is, the last constraint in  $\gamma$  describes the last state of a state sequence. For instance,  $\Delta(C_1 \mapsto C_2 \Rightarrow C_3 \searrow C_4) = 3$  since  $C_2$  and  $C_3$  describe possibly the same state and  $C_4$  must describe a visible state.

$\Delta$  is simplified considerably for the case of a cartoon. Furthermore, since  $\Delta$  has only to be computed for voters and not any cartoon instance,  $\Delta$  reduces to 1 as will be shown in Section A.4.4.

This can also be seen by looking at the definitions of the temporal operators. The languages associated with the operators are  $vv$  for  $\mapsto$ ,  $vv^*$  for  $\Rightarrow$ , and  $vu^*(v \cup u)$  for  $\searrow$ . The  $v^*$  can be ignored because  $\Delta$  returns the minimal length. For  $C_1 \mapsto C_2 \Rightarrow C_3 \searrow C_4$  we therefore get the minimal language  $vv(v \cup u)$ , which contains at least three  $v$ 's. Therefore,  $\Delta(C_1 \mapsto C_2 \Rightarrow C_3 \searrow C_4)$  must be three, also.

The dynamic preference is defined such that a cartoon instance is preferred if it describes a state sequence that starts earlier or, with lower priority, ends later.

**DYNAMIC PREFERENCE RELATION  $\succ_d$ :** Dynamic preference is the relation that a cartoon instance  $\alpha^\sigma$  is preferred over another cartoon instance  $\beta^\mu$ ; that is,  $\alpha^\sigma \succ_d \beta^\mu$ , is defined as

$$\begin{aligned} \alpha^\sigma \succ_d \beta^\mu &\equiv t(s_a^\alpha) < t(s_a^\beta) \vee \\ &\quad (t(s_a^\alpha) = t(s_a^\beta) \wedge \\ &\quad \Delta(C_{i_\alpha} \rightarrow \dots \rightarrow C_{k_\alpha}) > \Delta(C_{i_\beta} \rightarrow \dots \rightarrow C_{k_\beta})) \end{aligned}$$

The cartoon instances  $\alpha^\sigma$  and  $\beta^\mu$  are dynamically indifferent,  $\alpha^\sigma \sim_d \beta^\mu$ , if the state sequences both start and end at the same time.

$$\alpha^\sigma \sim_d \beta^\mu \equiv t(s_a^\alpha) = t(s_a^\beta) \wedge \Delta(C_{i_\alpha} \rightarrow \dots \rightarrow C_{k_\alpha}) = \Delta(C_{i_\beta} \rightarrow \dots \rightarrow C_{k_\beta})$$

Finally, the main preference relation can be defined as a function of the static and the dynamic preference.

**PREFERENCE RELATION  $\succ$ :** The preference relation  $\succ$  holds between any two cartoon instances  $\alpha^\sigma$  and  $\beta^\mu$  and consists of a static preference  $\succ_s$  and a dynamic preference  $\succ_d$ . A cartoon instance  $\alpha^\sigma$  is preferred over a cartoon instance  $\beta^\mu$  if  $\alpha^\sigma \succ \beta^\mu$ , which is defined as

$$\alpha^\sigma \succ \beta^\mu \equiv \alpha^\sigma \succ_s \beta^\mu \vee (\alpha^\sigma \sim_s \beta^\mu \wedge \alpha^\sigma \succ_d \beta^\mu)$$

Two cartoon instances are indifferent with respect to the preference relation  $\succ$  if neither of the instances is preferred over the other; that is,

$$\alpha^\sigma \sim \beta^\mu \equiv \neg(\alpha^\sigma \succ \beta^\mu \vee \beta^\mu \succ \alpha^\sigma)$$

### A.4.3 Discussion of the Preference Relations

The preference relation was defined to have the four necessary characteristics. The first two requirements, that the user overrides any dynamic preference scheme and has to order them only partially, are trivially satisfied. The other two, reiterated below, are discussed in more detail.

- (3) A cartoon instance is more important the longer its past is, that is, the more past states it describes.
- (4) A cartoon instance is more important the farther into the future it looks, that is, the more future states it describes.

Items (3) and (4) are implemented by the dynamic preference relation. Assume that  $t_a^\alpha$  is the time of the first state in cartoon instance  $\alpha^\sigma$  and  $t_z^\alpha$  is the time of the earliest last state as, described by function  $\Delta$  in Section A.4.2. Let  $t_a^\beta$  and  $t_z^\beta$  be the corresponding times for cartoon instance  $\beta^\mu$ , and finally, let  $t$  denote the time of the current state.

Assume that cartoons  $\alpha$  and  $\beta$  are statically indifferent, that is,  $\alpha \sim_s \beta$ . There are three cases to consider for the discussion of the dynamic preference relation  $\succ_d$ .

$t_a^\alpha < t_a^\beta$ : In this case, cartoon instance  $\alpha^\sigma$  is preferred over  $\beta^\mu$ . At time  $t_a^\alpha$  instance  $\alpha^\sigma$  was initialized.  $t_a^\beta - t_a^\alpha$  time steps later, instance  $\beta^\mu$  was created. There is no reason why  $\beta^\mu$  should interrupt a process described by  $\alpha^\sigma$ . An interruption would be acceptable only if  $\beta^\mu$  was *statically* preferred over  $\alpha^\sigma$ .

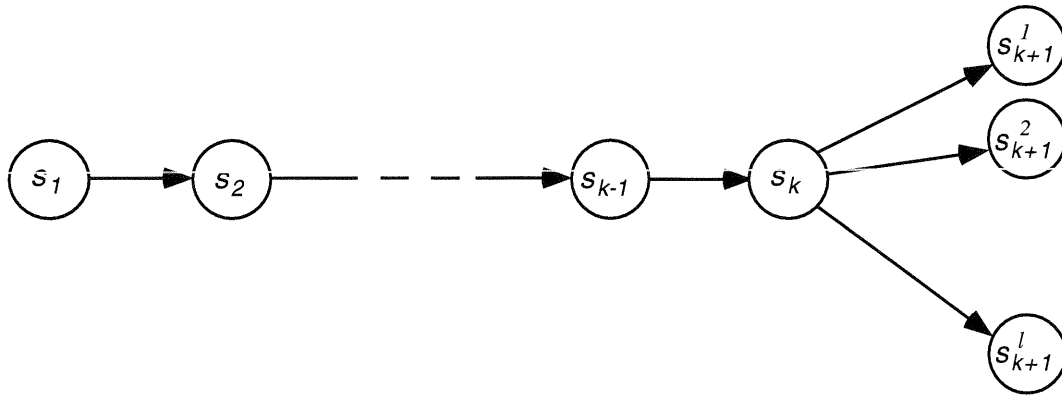
$t_a^\alpha = t_a^\beta \wedge t_z^\alpha > t_z^\beta$ : Again,  $\alpha^\sigma$  is preferred over  $\beta^\mu$ . A cartoon instance with a larger  $t_z$  is to be preferred because it promises to be of greater use in the future. Assume that  $t_z^\beta - t = 1$  and  $t_z^\alpha - \delta t > 1$ . After  $\beta^\mu$  has been used to reach the next state at time  $t + 1$ ,  $\beta^\mu$  is possibly thrown out and of no further use. This is not true for  $\alpha^\sigma$  because it still can be used for at least one more state. Furthermore, its history has grown and it therefore has become more important with respect to item (3) in the previous list.

$t_a^\alpha = t_a^\beta \wedge t_z^\alpha = t_z^\beta$ : Currently, there is no further tie breaker for this situation and the two cartoon instances are indifferent. In this case, a cartoon instance is selected at random.

#### A.4.4 Preference Relation for Cartoons

The only difference between cartoons and temporal constraints relevant to the preference relation is concerned with the last two constraints of a cartoon, which are always of the form  $C_k \searrow C_{k+1}$ .  $t_z - t = 1$  has the same value for every cartoon instance and can therefore be ignored. The definition of  $\alpha^\sigma \succ_d \beta^\mu$  reduces to  $t_a^\alpha < t_a^\beta$ , leading to following definition of the preference relation.

$$\alpha^\sigma \succ \beta^\mu \equiv \alpha \succ_s \beta \vee (\alpha \sim_d \beta \wedge t_a^\alpha < t_a^\beta)$$



**Figure A.7:** State tree described by the cartoon  $C_1 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$ . States  $s_1, \dots, s_k$  are consistent with constraints  $C_1, \dots, C_k$ , respectively, and states  $s_{k+1}^1, \dots, s_{k+1}^l$  are consistent with  $C_{k+1}$ .

A cartoon instance is preferable if the associated cartoon is preferable or, if this is not conclusive, if the instance is older. Two cartoons are indifferent if  $\alpha \sim_d \beta \wedge t_a^\alpha = t_a^\beta$ .

For a voter  $\alpha^\sigma$ ,  $t_a^\alpha$  is  $t - k_\alpha + 2$ , where  $k_\alpha$  is the number of constraints in  $\alpha = C_1 \rightarrow \dots \rightarrow C_{k_\alpha}$ . Thus, the preference relation between cartoon instances can be further simplified to

$$\alpha^\sigma \succ \beta^\mu \equiv \alpha \succ_s \beta \vee (\alpha \sim_d \beta \wedge k_\alpha > k_\beta)$$

This shows that the preference between two voters can be completely determined from their cartoons. This is not the case for the more general case of temporal constraints.

## A.5 Subsumption Hierarchy of Cartoons

Cartoons are sometimes preferred over each other, but this does not imply that they are not consistent with each other. Because a cartoon's second to last constraint  $C_k$  must describe parts of the current state  $s_k$ , a cartoon  $C_1 \rightarrow \dots \rightarrow C_k \rightarrow C_{k+1}$  describes a state tree of the form shown in Figure A.7.

A cartoon instance is a cartoon that is associated with a state sequence, which is described by an initial sequence of constraints of the cartoon. The states  $s_1, s_2, \dots, s_{k-1}$  have exactly one child, and state  $s_k$  has  $l$  children, where  $l$  is the number of successor states of  $s_k$  that are consistent with  $C_{k+1}$ . Of course,  $l$  is at most the number of actions possible in the current state  $s_k$ . Let  $\alpha^\sigma(s)$  be the set of children of  $s$  that are described by  $\alpha^{\sigma, \sigma'}$ , that is, by a specialization of the cartoon instance  $\alpha^\sigma$ .

**CONSISTENCY OF TWO CARTOON INSTANCES:** Two cartoon instances  $\alpha^\sigma$  and  $\beta^\mu$  are consistent in state  $s$  if they have at least one possible future state in common, that is, if

$\mathcal{I}(s, \alpha^\sigma, \beta^\mu)$  holds, which is defined as follows.

$$\mathcal{I}(s, \alpha^\sigma, \beta^\mu) \equiv \alpha^\sigma(s) \cap \beta^\mu(s) \neq \emptyset$$

This definition can now easily be extended to a set of cartoon instances by making sure that all the instances are pairwise consistent.

**CONSISTENCY OF A SET OF CARTOON INSTANCES:** A set  $I = \{\alpha_1^{\sigma_1}, \dots, \alpha_k^{\sigma_k}\}$  of cartoon instances is consistent in state  $s$ , if the instances are pairwise consistent, that is, if for each  $i, j$  such that  $1 \leq i < j \leq k$ ,  $\mathcal{I}(s, \alpha_i^{\sigma_i}, \alpha_j^{\sigma_j})$  holds. This is expressed by  $\mathcal{I}(s, I)$ .

Although the cartoon instances have to be pairwise consistent, testing a set for consistency can be done quite quickly if the intersection of all instances is stored. Assume that  $\mathcal{I}(s, \{\alpha_1^{\sigma_1}, \dots, \alpha_k^{\sigma_k}\})$  and that  $A$  is the intersection of all these instances, that is,  $A = \bigcap_{i=1}^k \alpha_i^{\sigma_i}(s)$ . Then,  $\mathcal{I}(s, \{\alpha_1^{\sigma_1}, \dots, \alpha_k^{\sigma_k}, \alpha_{k+1}^{\sigma_{k+1}}\})$  holds if  $A \cap \alpha_{k+1}^{\sigma_{k+1}} \neq \emptyset$ .

In general, it does not make sense to say that two cartoons are consistent because it depends on which past and present states they describe; that is, they depend on the variable binding. The consistency predicate  $\mathcal{I}$  can now be used to check whether cartoon instances can be subsumed.

Quite often several actions can be executed, leading to a state described by the same cartoon instance. Instead of picking one of the actions randomly, cartoon instances can be subsumed to reduce the number of possible actions and to improve the importance of some of the actions.

### A.5.1 Subsumption Conditions

Cartoon subsumption is based on the idea that more than one cartoon can suggest the same action. The action is chosen that gets the “best” votes. Then, one of the set of best actions is chosen randomly and with it the associated state, which will then become the new current state. To simplify the notation for cartoon instances,  $v_i$  will be used instead of  $\alpha_i^{\sigma_i}$ . The possible successors of the current state  $s$  are  $s_1, s_2, \dots, s_m$ . Pairs of the form  $\langle s_i; v_1, \dots, v_n \rangle$  are associated with each possible future state  $s_i$ , where the  $v_j$ 's are the cartoon instances with  $s_i$  as last state. A cartoon instance  $v_i$  is preferred over another instance  $v_j$  if  $v_i \succ v_j$ .

The concept of a *voter* is defined next. A voter is comparable to an instance of a rewrite rule. Whereas the rule instance is used to execute certain actions, a voter is used to vote for a certain state.

**VOTER:** A voter  $v$  is a cartoon instance in which the state  $s$  is consistent with the last constraint of the cartoon. It is said that  $v$  votes for  $s$ .

Several cartoon instances may vote for the same state. This group of cartoon instances is called the *supporter* of a state and is defined as follows.

**SUPPORTER:** A supporter of a state  $s$  is a pair of the form

$$\langle s; v_1, v_2, \dots, v_k \rangle,$$

where  $v_1, \dots, v_k$  is a list of a maximally consistent set of voters of  $s$ , sorted according to the preference relation  $\succ$ .

To state all the conditions on superimposed cartoon instances, the following notation is used.

$$r \equiv \langle s; v_1, v_2, \dots, v_k \rangle = \langle s; I \rangle$$

The voters in  $r$  are ordered according to their preferences. The conditions that hold on this order are then

$$\forall i, 1 \leq i < k. \neg(v_{i+1} \succ v_i)$$

or equivalently

$$\forall v_i, v_j \in I. v_i \succ v_j \supset i < j$$

A maximally consistent set of voters contains as many voters as possible under the constraint that all these voters must be pairwise consistent. The following definition defines the set by asserting that it is consistent and no other voter for the same state can be added to the set without making the set inconsistent.

**MAXIMALLY CONSISTENT SET OF VOTERS:** Let  $V$  be the set of all voters for state  $s$ . A consistent subset  $V'$  of  $V$  exists such that  $V' \subseteq V$  and  $\mathcal{I}(V')$ . The set  $V'$  is maximal with respect to  $V$  and  $\mathcal{I}$  if there is no other set  $V'' \subseteq V$  that is consistent and  $V' \subset V''$ . (Note that  $V' \subset V'' \supset V' \neq V''$ , where  $\supset$  stand for “implies.”) Formally,  $V'$  is a maximally consistent subset of  $V$  if

$$V' \subseteq V \wedge \mathcal{I}(V') \wedge \neg \exists V''. V'' \subseteq V \wedge \mathcal{I}(V'') \wedge V' \subset V''$$

Because all the voters for a state  $s$  vote for the same state  $s$ , they are all consistent. Therefore, the set of all voters for a state  $s$  is always maximally consistent.

Finally, the preference relation between voters can be extended to supporters. A supporter  $r_i$  is preferred over a supporter  $r_j$ , if  $r_i$ 's voters are preferred over  $r_j$ 's voters, that is, if  $r_i \succ r_j$ , which is defined by

$$\begin{aligned} r_i \succ r_j &\equiv v_1^i, \dots, v_h^i \succ v_1^j, \dots, v_k^j \\ &\equiv \exists l, l \leq h. \forall m, m < l. v_m^i \sim v_m^j \wedge (k < l \vee v_l^i \succ v_l^j) \end{aligned}$$

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  $l^{\text{th}}$  voter makes the difference:

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

Computing the states associated with the best supporters is simple, as shown by the function SUBSUMPTION in Figure A.8. It takes the current state  $s$  and the set  $A$  of possible actions in  $s$  as parameters and returns the set of best successor states of  $s$ . Two cartoon instances that end in one of the successor states of  $s$  are consistent if they end in the same state. Therefore, iterating over the successor states of  $s$  and collecting the supporters for each state assures that for each state the voters  $v_1, \dots, v_k$  are maximally consistent because all the cartoon instances are voters.

A possible extension of the voting mechanism could include the use of cartoon instances than are no voters (according to the current definition of a voter).

### A.5.2 Avoiding Combinatorial Explosion with Cartoon Subsumption

Given are two cartoons  $\alpha = C_1^\alpha \rightarrow C_2^\alpha$  and  $\beta = C_1^\beta \rightarrow C_2^\beta$ . Assume that  $C_1^\alpha \supset C_1^\beta$  and  $C_2^\alpha \supset C_2^\beta$ ; that is, the constraints in  $\alpha$  imply the constraints in  $\beta$ . Then  $\alpha$  will always suggest a subset of the actions that are suggested by  $\beta$ . Formally, this means

$$(C_1^\alpha \supset C_1^\beta \wedge C_2^\alpha \supset C_2^\beta) \supset \forall s. \forall \alpha^\sigma \text{ voting for } s. \exists \beta^\mu \text{ voting for } s, \text{ or}$$

$$(C_1^\alpha \supset C_1^\beta \wedge C_2^\alpha \supset C_2^\beta) \supset \forall s. \forall \alpha^\sigma \exists \beta^\mu. \alpha^\sigma(s) \subseteq \beta^\mu(s)$$

```

SUBSUMPTION( $s, A$ )
  for  $a \in A$  do
     $s' \leftarrow a(s)$ 
     $R \leftarrow R \cup \{\langle s'; v_1, \dots, v_k \rangle\}$ , the  $v_1, \dots, v_k$  are all the voters for  $s'$ 
  end for
   $R' \leftarrow \{r \mid r \in R \wedge \neg \exists r'. r' \in R \wedge r' \succ r\}$ 
   $S \leftarrow \{s' \mid \langle s'; I \rangle \in R'\}$ 
  return  $S$ 
end SUBSUMPTION

```

**Figure A.8:** The SUBSUMPTION function returns the set of best states that can be reached by one action in  $A$  from  $s$ .

This means that  $\alpha$  is a special case of  $\beta$ . If a state gets a vote from  $\alpha$  then it will certainly also get one from  $\beta$ , thus getting two votes. Therefore, a state that gets a vote from  $\alpha$  has a better chance of being chosen than one that gets a vote from  $\beta$  only.

This effect can also be accomplished in a rule-based system with rule ordering. However, using cartoons opens the door to more possibilities. First, if there are just pairs of cartoons that interact with each other, as shown above, no rule ordering is necessary. Second, choosing the state that gets the most votes is not possible in rule-based systems but can be of great value in some cases.

Let  $\mathcal{R}$  be the set of rules  $\{r_1, \dots, r_m\}$  and let  $\mathcal{C}$  be the set of cartoons  $\{\alpha_1, \dots, \alpha_n\}$ , where  $\alpha_1 = C_{i,1} \rightarrow C_{i,2}$ . There are  $2^n$  possible conditions because the conjunction of the conditions in any subset of  $\{C_{1,1}, \dots, C_{n,1}\}$  is a potential condition. Furthermore, each combination can choose from up to  $m$  actions. Therefore,  $m$  rules and  $n$  cartoons can require up to  $m \cdot 2^n$  rules in a purely rule-based system. However, this case is not to be expected in practical applications.

## A.6 Summary

In the subsumption-like architecture of Cartoonist, cartoons can be combined to describe complex behaviors. Although it makes sense to think of Cartoonist's subsumption mechanism as combining the cartoons, the realization of this idea is done rather different. Cartoons vote for certain states. The voters are organized into consistent groups. The state with the best support, that is, with the best group of voters, is selected.



Voters generated by cartoons describe state sequences that differ in the last state only, which makes finding the maximally consistent sets simple. This is a further advantage of the very definition of a cartoon.