Automatically Improving Agents Behaviors in an Urban Simulation

Tristan Cazenave
LIP6
Tour 46-00 2ème étage
Université Pierre et Marie Curie
4, place Jussieu, 75252 Paris Cedex 05, France
e-mail: Tristan.Cazenave@poleia.lip6.fr

Abstract

Our goal is to create realistic urban simulations involving pedestrians, cars, pedestrians crossings and many others urban agents. These simulations help architectural designers in choosing architectural configurations. A problem related to this simulation is to create agents that have realistic behaviors and that are also efficient (a simulation may manage thousands of agents at the same time, so modeling an agent's behavior has to be rapid). Therefore, we have developed a program that automatically improves the agents behaviors given (1) some simple situations to avoid (a car that run over a pedestrian, or a pedestrian that tries to walk on another one) and (2) the rules of the simulation. The rules that describes the world and the rules describing the situations to avoid are written using predicate logic. The program that automatically writes the agents is written using metapredicates that manipulates the predicate logic rules describing the simulation. The advantage of creating the agents automatically is to have a lot of reliable, efficient and quickly designed rules.

Keywords

Automatic Program Synthesis, Multi Agents Urban Simulation.

1 Introduction

Our goal is to create realistic urban simulations involving pedestrians, cars, pedestrians crossings and many others urban agents. These simulations help architectural designers in choosing architectural configurations. A problem related to this simulation is to create agents that have realistic behaviors and that are also efficient (a simulation may manage thousands of agents at the same time, so modeling an agent's behavior has to be rapid). Creating a realistic agent's behavior manually is hard because of the great number of cases and interactions that can take place. Two programmers have worked on programming manually agents behaviors during two months, but some of the agents still had unrealistic behaviors, leading to unrealistic simulations. Moreover, the model was very sensitive to changes in an agent: a little and apparently unimportant change in an agent could transform a working simulation into an unrealistic one. Therefore, we have developed a program that automatically improves the agents behaviors given (1) some simple situations to avoid (a car that run over a pedestrian, or a pedestrian that tries to walk on another one) and (2) the rules of the simulation. The rules that describes the world and the rules describing the situations to avoid are written using predicate logic. The program that automatically writes the agents is written using metapredicates that manipulates the predicate logic rules describing the

simulation. The metarules are in charge of writing all the possible rules that can lead to a situation to avoid in the next steps of the simulation. This enables the agents using these rules to be more realistic. The advantage of creating them automatically is to have a lot of reliable, efficient and quickly designed rules. The creation of all the rules is made by replacing some predicates in the rules that describe the situations to avoid, with their definitions contained in the rules of the simulation. Our approach to automatic agent improvement is efficient and can be used in other contexts.

In a first part, I describe how programs are automatically written by my system. In a second part, I explain the application to the urban simulation. Some earlier work on this system can be found in [1]. This type of automatic program improvement has been first formalized in [2,5]. It was latter empirically tested in [3,4]. Our main domain of research is metaknowledge [6], and especially the automatic creation of efficient and reliable programs from a simple and declarative description.

2 Automatic Program Synthesis

In this section, we begin with the input and output of the system. Then we describe how the output is calculated given the input.

2.1 Input of the system

The system that creates the agents is given four types of rules: rules describing the simulation, rules about the goals to achieve, metarules about the monovaluation of some predicates, metarules about the impossibility of some other rules.

All these rules are expressed in predicate logic. Metarules use some metapredicates so as to create, transform and delete other rules and other predicates. For the sake of simplicity, we give the rules in a Prolog like formalism even if this is not exactly the way they are represented in our system.

Example of a rule describing the simulation:

```
Vision_angle ( ?n 1 ) :-  Position_pedestrian ( ?n1 ?x ?y ) , Dx_angle ( ?n ?dx ) , Dy_angle ( ?n ?dy ) ,
                          equal ( ?x1 add ( ?x ?dx ) ) , equal ( ?y1 add ( ?y ?dy ) ) ,
                          Identification_case ( ?n2 ?x1 ?y1 ) , not_equal ( ?n2 -1 ) , not_equal ( ?n1 ?n2 ).
```

This rule means that the emplacement that is one step ahead of the pedestrian with angle ?n*PI/10 cannot be occupied by the pedestrian (Vision_angle (?n 1)). This is due to the fact that the position of the pedestrian number ?n1 is at location ?x,?y (symbols with question marks are variables), and that a step in the direction of angle ?n*PI/10 would make him move at ?x+?dx,?y+?dy. Unfortunately, the number (?n2) of the emplacement at ?x+?dx,?y+?dy is not empty (not equal to -1) and not already occupied by the pedestrian (not equal to ?n1).

There are sixty rules that calculate all the predicates related to the choice of the orientation of the pedestrian in the simulation.

Example of a rule about the goal to achieve:

```
Delta_end ( ?dx ?dy ) :-  Vision_angle ( ?n 0 ) , All_smaller_angles_impossible ( ?n ) ,
                          greater_than ( ?n -1 ) , greater_than ( 10 ?n ) ,  Dy_angle ( ?n ?dy ) ,
                          Dx_angle ( ?n ?dx ).
```

Delta_end (?dx ?dy) is the final move that is chosen by the pedestrian. This final move corresponds to the angle ?n*PI/10 if all angles smaller than ?n*PI/10 lead to impossible moves and if ?n is between -1

and 10. The goal of the method that our system optimizes is to find the move of each pedestrian in the simulation. It is called very often and it is a time consuming method.

There are two rules about the goal to achieve in the urban simulation application.

Example of a metarule about the monovaluation of a predicate:

```
replace_variable ( ?r ?var1 ?var4 ) :-    rule  ( ?r ) ,
                                          condition ( ?r Identification_case ( ?var1 ?var2 ?var3 ) ) ,
                                          condition ( ?r Identification_case ( ?var4 ?var2 ?var3 ) ) ,
                                          not_the_same ( ?var1 ?var4 ).
```

This rule means that there is only one possible value for each emplacement in the simulation. If the system creates a rule that contains two different variables for the same emplacement, then it replaces one of the variables by the other one (?r is a variable containing a rule, ?var is a metavariable containing another variable, the metapredicate 'condition' looks for all the conditions in rule ?r that match the given predicate).

There are nine rules about the monovaluation of a predicate.

Example of a metarule about the impossibility of another rule:

```
remove_rule ( ?r )  :-    rule ( ?r ) , condition ( ?r equal ( ?var ?var1 ) ) , constant ( ?var ) ,
                          constant ( ?var1 ) , not_the_same ( ?var ?var1 ).
```

This rule means that if two constants must be equal and are not the same, then the rule can never be applied. So the system removes it.

There are five rules about the impossibility of other rules.

2.2 Output of the system

A simplified output of the system is the following C++ method that tells if a pedestrian can make a move, and where he will be located at the end of the move (dX and dY are the pedestrian's move coordinates).

```
int Pedestrian::Evite () {
reel=dX;
reel1=dY;
n=Id; reel2=X; reel3=Y;
reel4 = (((reel1) * (0.951057)) - ((reel) * (0.309017)));
reel5 = (reel3 + reel4);
reel6 = (((reel) * (0.951057)) + ((reel1) * (0.309017)));
reel7 = (reel2 + reel6);
n1= Identification_case (reel7,reel5);
if ((n1 == n) || (n1 == -1)) { dX=reel6; dY=reel4; return 1; }
return 0; }
```

The function returns 1 if the pedestrian can move, 0 else. The real C++ function created by our system is much longer than the one above. Note that 0.951057=cos (PI/10) and that 0.309017=sin (PI/10).

2.3 Program Synthesis

A program takes the rules defining the goals and repeats the specialization of the rules until no more specialization can be made. This formalized in the following pseudo-code algorithm:

```
while (some rules to specialize) {
        r = rule to be specialized
        set_of_rules = replace a predicate in r by its definitions, creating a new rule for each definition
        match the metarules about monovaluation on set_of_rules
        match the metarules about impossible rules on set_of_rules
        remove r from the set of rules to specialize
        add set_of_rules to the set of rules to specialize }
```

After all the specialized rules have been created, they are put together into a tree. This tree is the compiled into a C++ program that can be included as a part of the overall simulation.

3 Application to the improvement of an urban simulation

Our program synthesis system has been applied to the improvement of an urban simulation involving pedestrians. We first describe the aim of the tool for urban simulation and the problems encountered when developing it. Then we show how we overcome this problems using an automatic agent synthesis method.

3.1 Urban Simulation

The simulation tool that is optimized by our system has been used to choose the configuration of the 'Grand Stade de France' that is built for the next World Soccer Cup in France. It has also been used to test various urban configurations such as Rail Stations or Roads Configurations in a city. A problem encountered in these simulation is that simulating realistic agents behaviors is time consuming, especially in simulations containing thousands of agents. Another problem is that making agents more complicated and more realistic makes the maintaining of the program harder, and also make changes in the program difficult to handle.

The solution we found to overcome these two problems is to automatically create efficient and realistic agents from a declarative description of their behaviors.

3.2 Agents Synthesis

The goal of the method that our system optimizes is to find the move of each pedestrian in the simulation. It is called very often and it is a time consuming method.

Given the rules presented in subsection 2.1, our system wrote a 5 Kilo Octets C++ method that is much faster than the original and equivalent hand coded C++ method.

The rapidity of the synthesized program is one advantage over the traditional programming approach. Another advantage is that it is easier to modify the behavior of an agent when it is written in a declarative logic language than when it is directly written in C++.

4 Conclusion

I have described a method to automatically create efficient programs given a declarative representation of a domain. This method has been successfully applied to automatically improve some agents behaviors in an urban simulation.

The C++ program written by our system is 10 times faster than the original hand-coded C++ program written by professional programmers. The main reason for the success of this approach is that hand-coded programs have to be maintainable and simple so that the programmer can understand them, whereas our system does not have this limitation. The clarity of an hand-made program is sometime at the price of its efficiency. Our system writes long and unclear programs, but they are faster than hand-coded programs because all the calculation that can be made at compilation time have been made.

Meanwhile, the agents can be modified more easily than in an hand-coded C++ program, because they are represented declaratively in a logic program.

Thus, our approach enables to write faster agents simulations, and also enables to modify agents behaviors in an easier way than by directly modifying the C++ code of the agent.

The method has a wide range of applications, especially in optimizing some time consuming simulations. In the near future, we plan to apply our system to other domains where time optimization leads to better results.

References

[1] Cazenave T. (1996). *Learning to Manage a Firm.* First International Conference on Industrial Engineering Applications and Practice, San Diego, 1996.

[2] Dejong G., Mooney R. (1986). *Explanation Based Learning : an alternative view.* Machine Learning 2, 1986.

[3] S. Minton, J. Carbonell, C. Knoblock, D. Kuokka, O. Etzioni, Y. Gil (1989). *Explanation-Based Learning : A Problem Solving Perspective*. Artificial Intelligence 40, 1989.

[4] S. Minton (1990). *Quantitative Results Concerning the Utility of Explanation-Based Learning*. Artificial Intelligence 42, 1990.

[5] Mitchell T. M., Keller R. M., Kedar-Kabelli S. T. (1986). *Explanation-based Generalization : A unifying view*. Machine Learning 1 (1), 1986.

[6] Pitrat J. (1990). *Métaconnaissances*. Hermès, 1990.