A Language for Geometric Reasoning in Mobile Robots

Joseph J. Pfeiffer, Jr.
Department of Computer Science
New Mexico State University
Las Cruces, NM, USA 88003
pfeiffer@cs.nmsu.edu

Abstract

Isaac is a rule-based language for mobile robots currently under development at NMSU. A successor to Altaira, it replaces Altaira's state-based rules and tile-based navigation with a more general geometric reasoning mechanism. The language uses the FuzzyCLIPS expert system shell as a reasoning backend.

Keywords

Rule-based languages, reasoning, robotics, fuzzy logic

A Language for Geometric Reasoning in Mobile Robots

Joseph J. Pfeiffer, Jr.
Department of Computer Science
New Mexico State University
Las Cruces, NM, USA 88003
pfeiffer@cs.nmsu.edu

Abstract

Isaac is a rule-based language for mobile robots currently under development at NMSU. A successor to Altaira, it replaces Altaira's state-based rules and tile-based navigation with a more general geometric reasoning mechanism. The language uses the FuzzyCLIPS expert system shell as a reasoning backend.

Introduction

Geometric reasoning for mobile robots is a natural problem domain for visual languages, as the concepts used (polygons, physical motion, ranging and other sensor returns) are strongly visual. The robot's environment, the path to be followed, and the actions it should perform are all represented quite naturally in pictorial form.

This paper discusses Isaac (named for the late Isaac Asimov, who coined the term robotics and explored the ethical implications of robots in his fiction), a visual language for geometric reasoning in mobile robots.

Isaac is a rule-based language, with rules enabled by the presence of objects in the robot's environment. These objects may be known to be in the environment before the program begins execution, they may be representations of sensor readings, or they may be created as a result of rule firings. In addition, they may represent objects that are actually physically present (such as walls), or they may be abstract concepts (such as a path to follow).

Isaac maintains representations of the robot and its environment using a two dimensional geometric modelling library developed by the project team. FuzzyCLIPS is used as the reasoning backend.

The paper is organized as follows: following this introduction, we discuss past research into visual languages for mobile robots. After discussing the deficiencies of tile-based navigation, specifically as used by Altaira, we describe Isaac itself, including its geometric modeling system, its rules, and the mapping from Isaac's geometric concepts to FuzzyCLIPS facts and rules. Finally, we give some preliminary conclusions.

Prior Research

In recent years, there have been a number of visual languages developed for mobile robots. These languages have been intended for small robots with limited processing power (typically using a Motorola HC11 or similar microcontrollers) constructed from LEGO parts. Four examples of these languages are Altaira, LEGOsheets, RCX Code, and VSL. Except for VSL, all of these languages are targeted at novice users (in particular, children).

Altaira[1] is a purely visual language for LEGO mobile robots. It uses a tile-based navigation system, and implements a state machine based on both tile and robot states. On each execution cycle, the sensors are all sampled, and the combination of sensor, tile state, and robot state inputs is used to select new tile and robot states, perform navigational updates, and control the robot's actuators. Altaira was a finalist in the 1997 Visual Programming Challenge. Isaac may be regarded as a successor language to this language, so we will be describing it and its deficiencies in more detail later.

LEGOsheets[2] is a hybrid visual-textual language intended specifically for use with LEGO robots, as a way of

introducing programming concepts to children. The rules are tied to the actuators (so each actuator has effectively an independent rule set), and are textual. There is no notion of saved state or navigation. LEGOsheets deserves special mention as the language which inspired the Visual Programming Challenge, held in conjunction with VL'96. and VL'97.

RCX Code[3] is a visual language developed by LEGO for use with their MINDSTORMS robotics products. This is an event-driven language, in which changes to monitored sensor result in the execution of a series of commands in a control-flow model.

VSL[4] is a straightforward implementation of Brooks's subsumption architecture[5], implemented in Prograph[6]. The same authors have also developed VBBL[7].

A number of other, more general, visual languages have also been applied to small mobile robot control in the context of the Visual Programming Challenge, including COCOA[8], Formulate[9], and Prograph. COCOA also deserves special mention here, as it was also a finalist in the 1997 Visual Programming Challenge.

Deficiencies in Tile-Based Navigation for Mobile Robots

In order to motivate the design of Isaac, we describe the salient features of its predecessor, Altaira. Altaira's execution model is essentially that of a two dimensional Turing machine. The environment is divided into square tiles (corresponding to the LEGO road tiles on which the robot operates), each of which is in some state. This tile state is used to represent the type of the tile, its orientation, and the history of the robot's entrances to and exits from the tile for purposes of making later navigational decisions.

There is also a state associated with the robot itself. This state is used to represent subgoals to be accomplished by the robot such as ''turn left at intersection.''

Altaira's rules map the robot's sensor inputs, the robot state, and the current tile state to actuator outputs, new robot and tile states, and navigational commands which serve to maintain a notion of the robot's orientation and the tile it is located on. A rule can specify any number of these inputs, or it can replace some or all of them with wild card inputs.

Altaira's ruleset is hierarchical, with the hierarchy defined by whether rules are conditioned on state inputs or not, and on the number of sensors used in defining a rule. This is used to implement a subsumption architecture with higher-level behaviors overriding lower-level behaviors (rules with robot state inputs define higher-level behaviors than rules with tile state inputs, and rules with tile state inputs define higher-level behaviors than rules that fail to specify either robot or tile state on input) and more specific rules (rules that specify more sensor inputs) overriding less specific rules (rules that specify fewer sensor inputs).

While Altaira's tile- and state-based model is appropriate for the limited environment for which it was intended, it is not adequate for a more general environment, for three primary reasons.

First, the tile-based navigation is itself inadequate. It assumes that the world can be divided into squares, and that it is relatively simple to determine when the robot has crossed from one to another. While this is true in some domains, it is in general too limiting.

Second, the use of a single monolithic state to represent a wide variety of different types of information relating to a tile leads to an explosion in both the number of states and the number of rules to deal with them.

Third, the language is only able to treat sensor inputs as boolean values. Support for sensors that return inherently scalar data (such as direction or range sensors) is not available in the language, nor is it clear how it could be added.

Work is continuing on addressing the first two issues within the context of Altaira, as the language's execution model remains well-suited to computing environments with limited processing power. At the same time, we have been exploring Isaac, a completely new language

intended to directly address these issues by eliminating the tile-based navigation and state-based reasoning with more general mechanisms. Isaac is intended to advance rule-based visual programming for robots to more general navigation, more powerful robots, and more sophisticated users.

Isaac

The Isaac environment includes two dimensional geometric environment and robot representations, editors for creating geometric models and rules, and a reasoning backend driven by FuzzyCLIPS[10].

Geometric Models

The geometric representation is a two dimensional modelling system using triangulated polygons. The system is hierarchical, allowing the representation of articulated objects. The geometric hierarchy is shown in Figure 1.

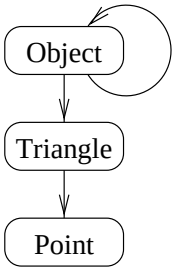

Figure 1: Geometric Hierarchy

An object is made up of an arbitrary number of triangles, and may also have subobjects. Each triangle is made up of three points.

Objects (including subobjects) have transformation matrices associated with them, so they can be translated or rotated by manipulating the matrices. As an object's hierarchy is traversed, the transformation matrices are multiplied so subobjects are located relative to their parents. This supports articulated objects, in which the subobjects are able to move independently of one another.

Objects also have sets of colors associated with them (so the object can be regarded as being of more than one color), for use in enabling rules for firing.

Typically, we will have two top-level objects: the robot and its environment. Figure 2 is an example of a possible environment.

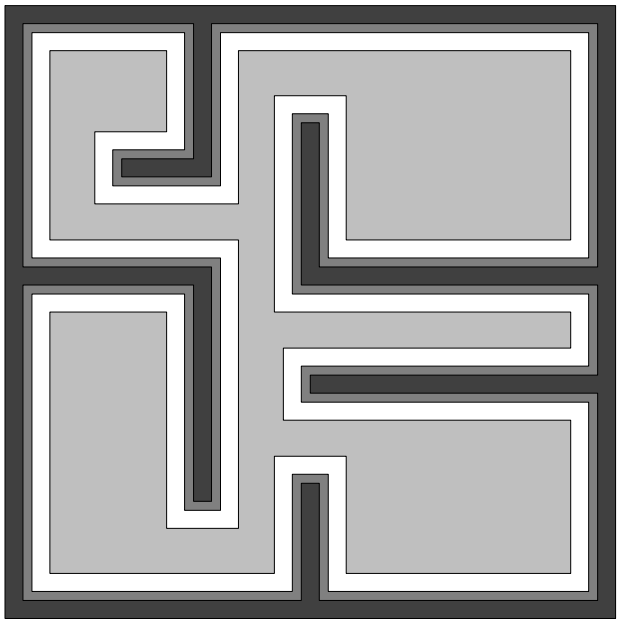

Figure 2: Example Robot Environment in Isaac

For the sake of clarity, the triangulation is not shown in Figure 2. The various objects in the environment are given different colors, to sensitize them for different rules. The meaning of the colors is given in Figure 3 (in the actual environment, these regions are assigned different colors by the user. As this paper is destined for printing in black and white, the regions are being shown in shades of grey, instead).

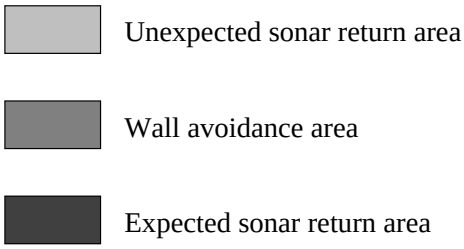

Figure 3: Interpretation of Region Colors

The dark regions represent areas from which sonar returns are expected (walls in the environment). Sonar returns from these areas result in manipulation of the robot's position.

The lighter areas represent collision avoidance areas. If the robot encounters these areas it is too close to the walls, and avoidance rules must be enabled.

The most lightly shaded areas represent unexpected sonar return areas. Sonar returns from these areas represent unexpected obstacles which must be added to the environment and avoided, as shown in the next section.

The unshaded areas will not enable any rules.

Rules

Rules take the form of a left hand side describing a situation which can occur, and a right hand side. When the left hand side situation is recognized, the rule is fired.

A rule can cause any or all of the following actions to take place:

1. Objects can be added to the environment.
2. Objects can be removed from the environment.
3. Geometric transformations can be applied to objects in the environment.

An example of a rule that inserts one object in the environment, and deletes another, is shown in Figure 4.

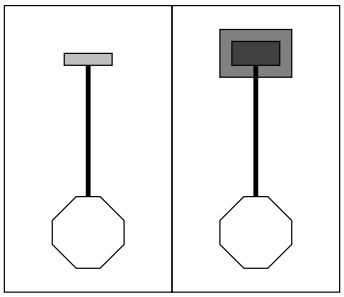

Figure 4: Rule for Obstacle Detection

This rule responds to a situation in which a sonar return indicates an unexpected object in the environment. The octagon is the object representing the robot; the sonar return is represented by the small box. As a result of the rule firing, the sonar return box is removed, and two new boxes, in different colors, representing the detected obstacle, are inserted. The inner box is a representation of the obstacle itself; the outer box is a new area from which sonar returns are now expected. In the event of a sonar return from the area around the box, the box will be modified to reflect the new information about its size. This does assume that the sonar return object has at least two members in its color set, so it can trigger rules from both expected sonar return areas and unexpected areas. The solid black line is a measuring stick: it states that the new obstacle should appear in the same position relative to the robot as the sonar return object.

Inputs

An input from any sensor results in an object being added to the robot model. On each execution cycle, all of the inputs are polled and objects added as appropriate. Typically, the rules which are fired by the presence of these objects will also remove them from the environment, in order to avoid polluting future execution cycles. The location and color of the object in relation to the robot is determined by the user, with the robot editor. Isaac supports four types of input devices:

Boolean sensors are devices such as touch sensors which provide a boolean value. On each execution cycle, a fixed-size object may be added to the environment depending on the state of the boolean sensor. The user may select to insert objects when the sensor returns true, when it returns or false, or different objects may be inserted on each condition.

Local analog sensors are devices such as light or flame sensors. The area of the object added as a result of the analog sensor is determined by the sensor reading.

Direction sensors are devices such as compasses. A direction sensor adds a fixed-size object a fixed distance from the center of the robot, at an angle to the centerline of the robot determined by the compass reading.

Ranging sensors are devices such as sonar or infrared ranging modules which return a value determined by the distance from the robot to a reflector. Ranging sensor values result in a fixed-size object being added to the environment at a fixed angle to the centerline of the robot,

at a distance determined by the value returned by the ranging sensor.

In the interests of simplicity in the programming model, there is no support as yet for imaging sensors (such as vision).

Outputs

Four types of actuators are supported. Actuators are subclassed from objects, so they can be added to or removed from the robot by the rules. When a rule adds or deletes an object representing an actuator, the corresponding robot actuator's state is controlled. There are two types of actuators:

Boolean actuators can only be given a value of true or false. A boolean actuator simply fires; its operation is analogous to the button on a spray can.

Analog actuators can be given a value from -1 to 1. Analog actuators are typically used in applications where speed control is needed.

Consideration is being given to adding linear and rotating servo actuators. If these are added, they will operate by adding an object of a given length or at a given rotation, and will cause the associated servo to track the length or rotation in the rule.

FuzzyCLIPS Backend

FuzzyCLIPS is an expert system shell based on CLIPS, enhanced to support fuzzy logic. Its operation is similar to CLIPS and other expert system shells, except that it is capable of handling fuzzy concepts and reasoning, and uncertainties in the rules and facts.

The shell extends on standard notions of expert systems by permitting assertions to include linguistic terms (such as asserting TEMPERATURE HOT), and permitting partial set membership. Rather than rules simply being enabled or disabled (as is the case in a conventional expert system), FuzzyCLIPS permits a rule to be enabled to the extent that its preconditions are satisfied. All rules are fired to whatever extent they are enabled, after which their results are combined in a defuzzification step.

An interface has been developed for manipulating the FuzzyCLIPS environment from C, so we will be able to make use of the FuzzyCLIPS reasoning engine from Isaac.

Execution Cycle

Isaac executes the following sequence of events on each execution cycle (the meanings of many of the terms used here will be made clear in subsequent sections).

1. Insert new objects in environment based on sensor inputs.
2. Determine object intersections, and assert facts in FuzzyCLIPS knowledge base.
3. Execute FuzzyCLIPS to obtain results of rule firings.
4. Apply results of rule firings to objects in geometric model.
5. Map output objects to robot actuators.

Mapping Geometric Relationships to FuzzyCLIPS Facts

Rules are enabled by detecting the presence of polygons in the robot model with specified colors in ''sensitized'' areas of the environment. Polygonal intersection is used to determine this, as follows:

On each execution cycle, each of polygons making up the representation of the robot is intersected with any polygons of the same color in the environment. For each non-empty intersection that is located, a fact is inserted in the FuzzyCLIPS database. The degree to which the fact is asserted is determined by the area of the intersection.

The fuzzy logic provided by FuzzyCLIPS is used to provide a very intuitive mechanism for sensor fusion: if several sensors return data, they may all enable different (and possibly conflicting) rules. Each of these rules fires, with the levels of confidence given by the intersection area. A decision is made regarding the final outcome by the rule

combining and defuzzification mechanisms of Fuzzy-CLIPS.

Editors

Editors are under development for geometric models and for rules. The geometric model editor permits the user to add and delete objects and triangles, and to position them relative to one another. Shorthands exist to add arbitrarily shaped polygons, which are triangulated as they are added to the model.

The robot editor is enhanced relative to the environment editor, as it can also be used to define sensor and actuator objects.

The rule editor is also enhanced relative to the environment editor, as it is also able to specify that geometric transformations be applied to objects in the environment and the robot.

Preliminary Conclusions

Isaac is being developed as a generalization of the concepts present in tile-based languages such as COCOA and Altaira. The environment is no longer broken into tiles, and the tile state (intended to abstractly represent features of the environment) is replaced by a direct representation of this environment. The robot state and sensor inputs are replaced by a geometric robot representation, in which sensor readings are directly placed in the representation for processing by the reasoning engine. The state changes, navigation commands, and actuator outputs from the rules are replaced by rules which directly add and remove objects from the environment, perform geometric operations on the robot, and control actuators through a consistent interface. The hierarchical ruleset is replaced through a fuzzy logic rule combining mechanism.

It is apparent at this point that Isaac is a substantial generalization of the concepts present in tile-, state- and rule-based visual languages. It remains to be seen whether it will be as effective at mobile robot control in general environments as these languages were in the LEGO environment.

References

1. Pfeiffer, J.J.Jr. (1998) ‘‘Altaira: a rule-based visual language for small mobile robots,’’ in *Journal of Visual Languages & Computing* 9(2), pp 127-150.
2. Gindling, J., A. Ioannidou, J. O. Lokkebo, A. Repenning (1995) ‘‘LEGOsheets: a rule-based programming, simulation and manipulation environment for the LEGO programmable brick,’’ in *Proceedings of the 11th IEEE Symposium on Visual Languages, pp 172-179.*
3. LEGO Group (1998) *LEGO MINDSTORMS*, `http://www.legomindstorms.com/`
4. Cox, P.T, T.J. Smedley, J. Garden, M. McManus (1997) ‘‘Experiences with visual programming in a specific domain − visual programming challenge ’96,’’ in *Proceedings of the 1997 IEEE Symposium on Visual Langauges*, pp 258-263.
5. Brooks, R.A. (1986) ‘‘A robust layered control system for a mobile robot,’’ in *IEEE Journal of Robotics and Automation* RA-2, pp 14-23.
6. Cox, P.T., F.R. Giles, T. Pietrzykowski (1989) ‘‘Prograph: a step towards liberating programming from textual conditioning,’’ in *Proceedings of the 1989 IEEE Workshop on Visual Programming*, pp 150-156.
7. Cox, P.T, C.C. Risley, T.J. Smedley (1998) ‘‘Toward concrete representation in visual languages for robot control,’’ in *Journal of Visual Languages & Computing* 9(2), pp 211-240.
8. Heger, N., A. Cypher, D.C. Smith (1998) ‘‘Cocoa at the Visual Programming Challenge 1997,’’ in *Journal of Visual Languages & Computing* 9(2), pp 151-170
9. Ambler, A. and A. Broman (1998) ‘‘Formulate solution to the Visual Programming Challenge,’’ in *Journal of Visual Languages & Computing* 9(2), pp 171-210.
10. Orchard, R.A. (1998) *FuzzyCLIPS Version 6.04A User’s Guide.*