A Common Framework for Input, Processing, and Output in a Rule−based Visual Language

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

Rick L. Vinyard, Jr.
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
`rvinyard@cs.nmsu.edu`

Bernardo Margolis
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
`bemargol@cs.nmsu.edu`

ABSTRACT

Isaac is a programing language for geometric reasoning intended for controlling mobile robots, currently under development at NMSU. Due to the application area, handling input and output (in particular, sensor input and actuator output) in a manner consistent with the language as a whole is particularly important in this language.

In this paper, we discuss the mechanisms employed by Isaac to provide a common framework for input, processing, and output in this language.

KEYWORDS

Rule−based visual languages, mobile robots, geometric reasoning

1. Introduction

In developing a visual language for a specialized problem domain, the characteristics of the language must be carefully matched to the requirements of the domain.

In the case of Isaac, a rule−based visual language for reactive robot control applications, the nature of the processing model (described below) makes the mechanism used for specifying I/O operations as important as any of the mechanisms used for specifying any of the other functionality of the language. In developing the Isaac I/O system, a number of requirements were established in order to meet this goal:

- Input and output functions should be configurable by the robot programmer.
- Input and output should make use of the same abstractions as those used for geometric reasoning.
- The editors used for specifying the mappings from input and output to the language's I/O abstractions should be as similar as possible to those used in the remainder of the language.
- While a certain amount of textual code will be required to provide low−level interfacing with the device drivers, this wrapper should be very thin, in the sense that the mappings mentioned in the previous item should define the I/O interface as completely as possible. This will help to make the system as flexible as possible, and make adding new devices as simple as possible.

The general approach to I/O in this work is to model sensor inputs as the firing of rules similar in form to the other rules used by the system, and to use the firing of designated rules to cause actuator outputs to occur. Each sensor on the robot is mapped to a rule adding input objects to the robot's world model; affine transformations are used to control the detailed placement of the resulting object. Similarly, when rules are fired which place output objects in the world model, an affine transformation is used to map the location of the output object to the specific value to be output to a particular actuator.

In this paper, we describe the I/O facilities which have been defined for Isaac, and their relationship with the other components of the language. The paper is organized as follows. Following this Introduction, Section 2 will describe some relevant background including a description of reactive mobile robot control and of the I/O facilities provided by selected visual languages which have been used or developed for mobile robots, and Section 3 gives a brief description of Isaac. Section 4 will describe the visual editors used in Isaac paying particular attention to the relationship between the I/O editor and the other editors. Section 5 will introduce the device interaction library, and Section 6 will present some preliminary conclusions and directions for future research.

2. Background

In this section, we introduce some relevant background, giving some of the motivations for the design of Isaac. This will include a brief description of reactive robot control, and mention some related visual robot control languages.

2.1. Reactive Control of Mobile Robots

Robot applications may be broadly divided into two classes: industrial robots, in which a stationary robot functions as an intelligent tool in the tightly constrained environment of the assembly line, and autonomous mobile robots, in which a mobile robot maneuvers in much less constrained environments such as offices, hallways, or even the surface of another planet.

These two very different environments call for radically different approaches, methodology, and even languages. For an industrial robot, it is appropriate to use a procedural methodology to control the robot's motion, with (possibly) fine adjustments based on sensor inputs. The robot may be programmed explicitly, using a procedural robot control language, or implicitly using a pendant to guide the robot through its sequence of operations.

In a mobile robot, a procedural programming approach becomes very difficult to employ successfully, since there is a much higher probability of reaching unanticipated situations. Because of this, a reactive control strategy (*i.e.* a control strategy based on ''reactions'' to the environment, using rules to combine sensor inputs with a world model to produce actuator outputs and changes to the world model) is a common and successful approach[2, 4, 6, 9, 10].

2.2. I/O in Rule−Based Visual Languages for Robots

Over the last several years, a number of visual languages have either been developed or adapted for robot control, beginning with LEGOsheets[5]. The majority of the approaches we will describe here were developed for use in the 1997 Visual Programming challenge[1], so their mechanisms are tailored to the sensor and actuator set available in the robot used in the Challenge.

2.2.1. Altaira Altaira, the immediate predecessor to Isaac, is a language for control of small mobile robots using tile−based navigation[12]. As it is intended for an environment with very limited processing power and sensor configurations, its I/O facilities are correspondingly limited. It was originally developed specifically for the 1997 Visual Programming Challenge, so its I/O facilities are tailored to the robot used in the Challenge.

This robot had five light sensors and two touch sensors. The environment and problem was constrained so that the light sensors only had to be able to detect ''light'' and ''dark,'' and did not need to return an analog reading. The robot also had two drive motors as its only actuators; the motors could be programmed to go either forward or reverse.

In Altaira, sensor inputs are thresholded so they can be regarded as simply indicating the presence or absence of a condition in the robot's surroundings. On each rule execution cycle, all sensor inputs are read, and the thresholded inputs used (along with state information maintained by the language) in a match against the left hand side of all the rules in the ruleset to determine whether a given rule is to be enabled on a given cycle. A rule can explicitly ignore any or all of the sensor inputs (this is referred to as using `dontcare` inputs).

Actuators are limited to simple forward−off−reverse motors; control for all of the actuators (along with the state information used for rule activation) is updated on each execution cycle. Just as it is possible to use a `dontcare` on input, a rule can leave any or all actuators unchanged (this is referred to as using `nochange` outputs).

The integration of I/O with reasoning in Altaira is provided by simply having all rules consider input and generate output. While this yields a very complete integration, it does not scale well to robots with large numbers of inputs or outputs, or inputs which cannot conveniently be thresholded and treated as boolean values.

2.2.2. Cocoa Cocoa is a simulation tool for children, using rules to define the behavior of characters on a stage[14]. The language was adapted for use in the 1997 Visual Programming Challenge by adding I/O functionality appropriate to the robot used in the Challenge[7].

In Cocoa's case, the base language did not support I/O, so an extension to the language was used to translate the inputs to a boolean vector. The input and output processing performed using this vector is very similar to Altaira's, though it is possible to define rules in which input and output do not appear at all (this is equivalent to using all `dontcare`s for input and `nochange`s for output in Altaira, though the Cocoa mechanism is clearer as it explicitly uses no input and provides no output for these rules). Also, Cocoa implements speed control for the motors.

2.2.3. HDM−SDM A visual environment for programming mobile robots using a Hardware Definition Module (HDM) and Software Definition Module (SDM) is described in [3]. HDM generalizes the boolean input model used by Altaira and Cocoa by partitioning the input values into an arbitrary number of cases, and also generalizes the vector of sensor inputs used by those languages by permitting the user to define ''compound sensors'' based on vectors of inputs. A visual editor is used to define the mapping from sensor inputs to iconic representations.

A similar visual editor is used to define actuators, allowing the user to control actuator speed and direction, and to use several icons to represent a motor that is stopped, going forward, or going in reverse (to use their example).

HDM also provides a robot editor, allowing the programmer to define the locations of the sensors and actuators on the robot for purposes of simulation.

3. Isaac Processing Model

Isaac is a rule−based language for mobile robots. It uses a more general geometric model than the tile−based navigation present in all of the languages described above, replacing this with arbitrary two−dimensional positioning of objects. It also replaces the crisp rule enabling of those three languages with a fuzzy logic approach, using intersections between objects on the left hand side of rules and objects in the world model to enable rules. The area of an intersection is used to determine the extent to which a rule is enabled, with FuzzyCLIPS[11] used as a reasoning engine to drive the actual execution of the rules.

A rule in Isaac consists of a left hand side specifying a configuration of objects in the vicinity of the robot in the world model, and a right hand side specifying changes to the world model that occur when the rule is executed. This may include adding objects to the model, deleting objects from the model, or modifying the position of the robot in the model. A more complete description of Isaac's processing model may be found in [13].

Isaac's input and output is handled consistently with other processing. Sensor rules translate sensor input readings into objects which are placed in the world model, while actuator rules are enabled by the presence of objects in the world model and result in output to actuators.

4. Isaac Editors

The Isaac programming environment makes use of a total of four closely related editors:

- Object editor for defining libraries of objects for use in the other editors, and for defining initial world models.
- I/O editor creating mappings from sensor inputs to geometric objects to be placed in the world model, and from objects in the world model to actuator outputs.
- Robot editor for instantiating sensors and editors defined using their respective editors, and positioning them relative to the robot body.
- Rule editor for developing rules which draw inferences from the world model, and modify the world model as a result.

The editors all make use of a similar user interface with common features. Each of the editors can be used individually, so it is possible to create a library of sensors and actuators which may be instantiated on a variety of robots, or create a variety of maps on which a robot can be placed.

The following sections describe each of these editors in greater detail.

4.1. Object Editor

The simplest editor of the four is the object editor, which is able to define the properties of an object. This is used to create a library of objects for later use in both this and the other editors, and to create an initial world model. As shown in Figure 1, the object editor provides a hierarchy view and a map view. The robot programmer is able to add objects to and delete objects from the map, and may modify the position and size of the objects. Objects may be selected for deletion, modification, or to serve as the parent for sub−objects to be created, by use of either the hierarchy or the map view. The programmer can create objects either by defining shapes in this editor, or by selecting pre−defined objects from a library (the library of pre−defined objects is in turn created by this same editor, and saved to the library instead of saving as a map).

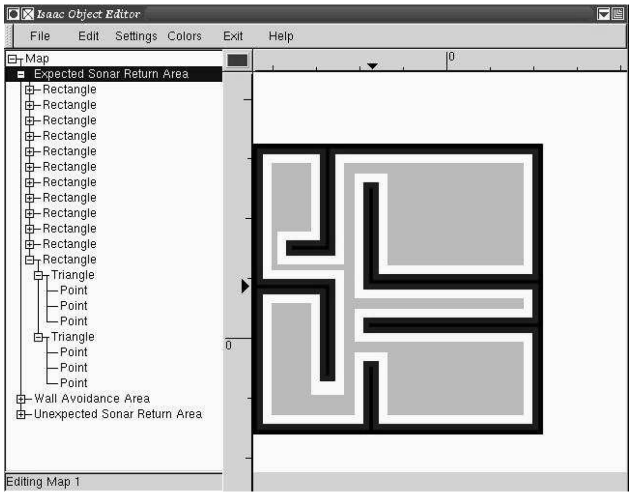


Figure 1. Object editor

The initial world model established in the object editor may be completely empty (as in the case of a totally unknown environment) or may contain an arbitrarily detailed map of the robot's expected surroundings. It is even possible to place ''objects'' in the world model which do not correspond to objects in the physical environment; this can be done (for instance) to represent a path to be followed by the robot.

As mentioned above, the object editor can also be used to create objects which can be placed in an object library, for use as predefined objects in the I/O, robot, and rule editors, as described below, or in defining maps.

4.2. I/O Editor

The I/O editor provides the programmer with a visual environment for defining a catalog of I/O objects of various types, and specifying the properties of these I/O objects. These properties include the specification of the actions necessary to insert an object into the world model in response to a sensor reading, and conversely, an actuator response to the presence of an object in the world model. The I/O object types defined in this editor can then be associated with specific sensors on the actual robot, as we will describe in Section 4.3.

An I/O rule is composed of the following:

- Sensor or Actuator Type
- Object
- Affine Transformation
- Sensor or Actuator Value

Sensor objects are created by selecting a sensor type from a catalog, defining a geometric object to place in the world model in response to a sensor input, and defining an affine transformation to apply to the object to modify its location or size due to the value of the sensor input.

Actuator objects are created similarly; in this case, the object will be recognized in the environment and the affine transformation is used to determine the value to be output to the actuator.

When the programmer begins definition of an I/O rule, a mnemonic name may be assigned to the rule. This helps to identify the rule in other editors such as the Robot Editor and Rule Editor.

The following subsections describe the use of the I/O editor in defining sensor and actuator objects. We will refer to Figure 2, a screen capture of the I/O editor, thoughout this description.

4.2.1. Sensor/Actuator Selection The programmer specifies the Sensor or Actuator component of an I/O object by selecting from a catalog of available sensors and actuators. This catalog is provided by the I/O editor in the form of a tree representing either an intuitive or transformational hierarchy as described in [15]. The catalog of sensors and actuators is shown in the left−most window in Figure 2. The tree structure shown provides the programmer with an intuitive hierarchy, and allows the programmer to select the sensor to be modified. In this example, a Polaroid L−series rangefinder transducer has been selected for this rule. The programmer could select another sensor for this rule by selecting another Polaroid series, a Murata sensor, or a completely different family of sensors.

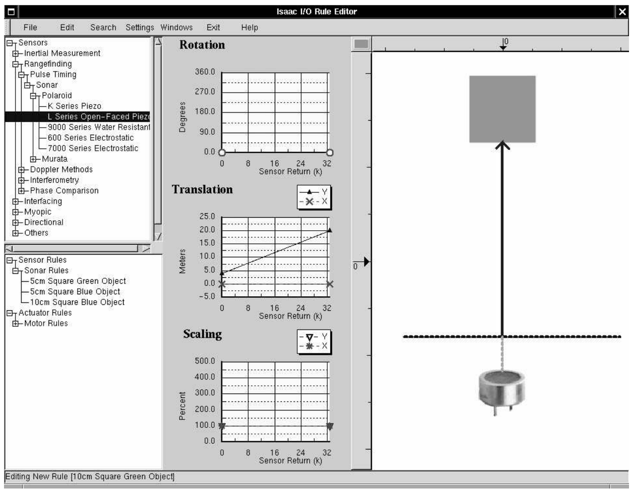


Figure 2. I/O editor

4.2.2. Object Definition The object to be associated with the sensor is defined in the I/O editor's map editor, shown in the right−most window of Figure 2, The programer is provided with two methods of modifying objects within the I/O editor. Double−clicking the object in the map window will provide the programmer with a catalog of all predefined objects, from which an object may be selected. Alternatively, the programmer is allowed to use the Object editor to create an entirely new object. For convenience, a quick color button is provided to the left of the horizontal ruler. This provides the programmer with a quick reference to the current color of the object. Clicking on this button provides the programmer with a color selection dialog allowing the color of the object to be changed without explicitly using the Object editor.

4.2.3. Transformation An affine transformation is the final programmer−specifiable component of an I/O object. The specified transformation will be applied to the selected object to provide a mapping from the original input to the position of an input object to an object in the world model, or conversely from the position the object in the world model to an actuator output.

Any of the transformations may be parameterized by the final component of an I/O rule (a sensor value or actuator value). Typically, the programmer will only parameterize a few of the transformations. The type of information returned by a sensor dictates which affine transformations are parameterized by the sensor value. Conversely, the type of information required by an actuator dictates which inverse transformations are parameterized by an actuator value(s). An example of a sensor type that would not parameterize an affine transform is a contact closure tactile sensor which has only boolean values. A sensor that would typically parameterize the affine transform is a sonar sensor which would apply its value to the Y component of the translation affine transform.

Thus, up to five functions may be specified by the programmer for each rule, as shown in Table 1.

Affine Transformation Component	***Functions***	
Rotation about the Z−axis	$R(sv)$	$R^{-1}(av)$
Translation along the X−axis	$T_X(sv)$	$T^{-1}_X(av)$
Translation along the Y−axis	$T_Y(sv)$	$T^{-1}_Y(av)$
Scaling in the X−axis	$S_X(sv)$	$S^{-1}_X(av)$
Scaling in the Y−axis	$S_Y(sv)$	$S^{-1}_Y(av)$

Table 1. Available affine transforms

The I/O editor provides the programmer with two methods of defining the transformation: either through manipulation of function plots specifying the affine transform component parameters, or through the map editor representing the location of the object in relation to the location of the sensor/actuator.

The three windows in the middle of Figure 2 comprise the plot view. The plot view allows the programmer to graphically edit the affine transformation component functions. In this example, the sonar sensor returns a value that may be used to determine the distance to an object. This value is used to parameterize the affine transform component function $T_Y(sv)$. The two scaling affine transforms $S_X(sv)$ and $S_Y(sv)$, the rotational transform $R(sv)$, and the translational affine transform $T_X(sv)$ are all constant functions. The object will not be rotated, will remain at its original size, and will not be translated along the X−axis, regardless of the sensor value.

In this example, the programmer has determined that a minimal sensor return of 0 will insert an object 4 meters away, and a maximal sensor reading of 32767 will insert the object 20 meters away. Thus, the programmer is specifying the following function:

$$T_Y(sv) = \frac{16m}{32767} * sv + 2m \simeq 0.49mm * sv + 2m$$

The map editor allows the programmer an alternative visual method of defining the affine transformation functions, without explicitly understanding affine transformations themselves. By dragging the horizontal dotted line up and down, the user defines the y−intercept of $T_Y(sv)$. By dragging the object in the map editor, the user defines the slope of $T_Y(sv)$. Thus, the user is provided with an intuitive method of programming the sensor.

Similarly, the map editor provides the programmer with a visual methodology for defining the other four affine transformation component functions. An interior bounding box is provided which allows the programmer to specify the minimum width and height of an object. Similarly, an exterior bounding box specifies the maximum width and height of an object. Values between the minimum and maximum are calculated by $S_X(sv)$ and $S_Y(sv)$.

A pair of rotational bars is provided to the programmer to similarly specify the minimum and maximum angular rotation values of a sensor.

The programmer can also vary the operation of the rule by changing the type of function associated with an affine transform component. Although only linear functions are mentioned above, the programmer could have chosen from a variety of functions including exponential, polynomial and logarithmic.

4.3. Robot Editor

The robot editor is used to define a mobile robot. Definition of the robot is a two step process which begins by creating a visual representation of the robot's physical configuration. The second step is to develop instantiation rules for the robot's sensors and actuators. This is done by assigning particular instances of I/O objects defined using the I/O editor to corresponding physical locations on the robot.

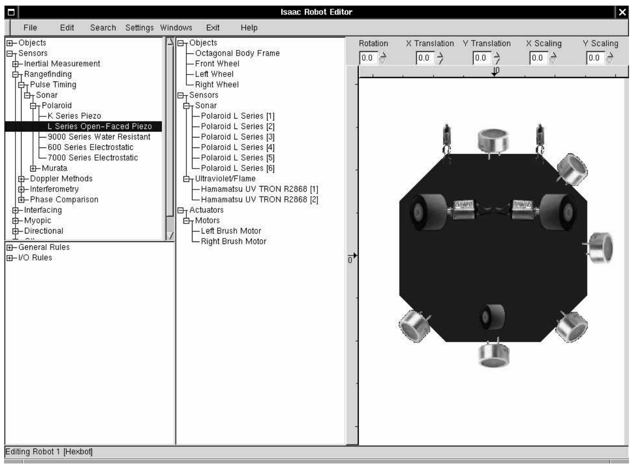
4.3.1. Physical Specification Creating a representation of the physical configuration of the robot is accomplished through several steps. The programmer is able to use any combination of objects, sensors and actuators to specify the components of the robot. A catalog of these components is presented in the leftmost upper window of Figure 3. The programmer selects these components by clicking on the desired component and dragging it to the map view (on the far right side of of the figure). This places the component on the robot, and allows the programmer to further define the instantiation of the component, if appropriate.

All components associated with the current robot are listed in the robot component window (middle window of Figure 3). Selecting a component in this window allows the programmer to change the properties of this component, or delete the component entirely.

Figure 3 shows the use of the robot editor to define a typical mobile robot with an octagonal body. The programmer has added six sonar rangefinders which will eventually comprise a ring of eight sonar rangefinders used by the robot to determine distance to obstacles.

Additionally, the robot has two ultraviolet sensors to determine the presence of a flame, an un−powered rear wheel for balance, two powered front wheels and two motors for movement.

Other components are necessary for a functional robot. These would include power supply, servo controller cards, and logic control mechanisms. If these components will not be associated with any rule, then it is at the discretion of the programmer to include or exclude these items from the physical description of the robot. In this example the three wheels are included as objects, but have no rules associated with them. However, if the wheels extruded from the frame of the robot, it is likely that collision avoidance rules would be developed for them.



Rotation 0.0 X Translation 0.0 Y Translation 0.0 X Scaling 0.0 Y Scaling 0.0

Editing Robot 1 [Hexbot]

Figure 3. Robot editor − physical specification

4.3.2. Component Instantiation When a component is instantiated, an affine transformation matrix is created which defines the component's location and orientation in reference to the robot's origin. For sensors and actuators an I/O rule is created, which performs the interaction between the I/O device and the world model.

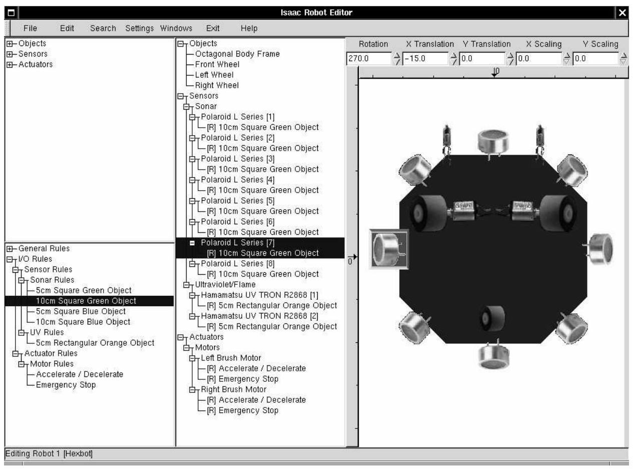
Defining the instantiation of a component requires that the programmer select the component in either the map view or the robot component window. The programmer is provided with two methods of specifying the affine transformation associated with a particular component. Using the map view, the programmer may drag the component into the proper position, specifying the translation component of the affine transformation matrix. Similarly, the programmer can use the rotation and scaling handles to rotate the component and scale it larger and smaller respectively.

Instead of using the map view to drag the sensor into the appropriate position, the programmer may use the spin buttons above the map view to specify the position, rotation and size of the component.

The body of the robot in Figure 4 is octagonal, with a maximal diameter of 30cm. The selected sonar sensor has been placed on the far left side of the robot, resulting in an X translation of −15cm from the center of the robot, and has been rotated 270˚ clockwise from the centerline of the robot.

The leftmost lower window of Figure 4 presents the programmer with all currently developed rules. To associate a component with a rule, the programmer drags the rule from the rule window and drops the rule on the component in either the map view or the robot component window. Multiple rules may be associated with a single component.

In this example, a sensor rule developed in the I/O editor, as shown in Figure 2, has been selected. This rule will obtain sensor readings from the sonar sensor on the left most side of the robot and insert objects in the world model as previously specified in the I/O editor after multiplying those objects by the affine transformation matrix of this sonar sensor. Similarly, an actuator rule maps transformations applied to the actuator object in a rule output to the device.



Rotation 270.0 X Translation -15.0 Y Translation 0.0 X Scaling 0.0 Y Scaling 0.0

Editing Robot 1 [Hexbot]

Figure 4. Robot editor − component instantiation

It is important to note that I/O objects may simultaneously be environmental objects as described in the Rule Editor. In this capacity, the I/O object is simply an object in the environment, and does not use the value of the I/O object to parameterize a rule. In Figure 4, the selected sonar sensor has been colored red. This coloration has no effect upon parameterized objects the sensor inserts into the environment. By coloring the sensor red, collision avoidance rules can be developed in the Rule Editor to prevent collisions involving extruding components. Since actual images of the sensors may be used by the programmer when desired, the coloration of a component is visually displayed to the programmer by placing a framed bounding box around the component when it is selected. The background color of the bounding box is the color of the object.

4.4. Rule Editor

The rule editor defines a robot's reactions to its world model. The left hand side of a rule is a configuration of one or more objects in the world model, while the right hand side represents the modifications which will be made to the world model if the rule is executed.

In operation, the left hand side editor is identical to the map editor, except that in addition to defining objects directly in this editor and selecting objects from the object library (described in section 4.1), the user can also select sensor objects defined in the I/O editor and instantiated with the robot editor.

Also, any modifications the user makes to the left hand side of the rule, except for insertion of sensor objects, is automatically made to the right hand side as well. This benefits both the programmer and the code generator by providing an interface in which initial conditions and modifications are explicitly (and separately) defined. Sensor objects are not echoed to the right hand side because they are intended to always be transitory, and are deleted as soon as they have been used to activate rules.

When an object is added to the left hand side of the rule, its origin is always the same as the robot's. Positioning the object defines a transformation, setting its new position relative to the robot's origin. For objects other than sensor objects the programmer has the freedom to modify the object by rotation, translation, or scaling; with sensor objects, the only modifiable parameter is that specified in the definition of the sensor object (so rule can be created that is responsive to sonar readings in a certain range, for instance).

The user is able to perform three types of modifications to the right hand side: objects may be added to the world model, they may be deleted from the world model, and translations and rotations may be applied to the robot icon. In addition to defining objects directly and using objects from the library defined using the map editor, it is also possible to insert actuator objects defined with the I/O editor.

Objects other than actuator objects added to the world model in the right hand side have their origins relative to the world (not relative to the robot, as is the case on the left hand side). The user, as with the left hand side, is able to apply transforms to set the location of the object.

In the case of actuator objects, the object location is relative to the robot, and the transformation applied to the actuator object will be used to control the output signal applied to the device. As with sensor objects, only the parameter defined in the I/O editor can be set with actuator objects. Unlike other objects, the actuator object is not actually added to the world model when its rule is fired; its usefulness is in generating the output.

The changes that are defined will be emphasized through the following visual means:

- when an object is removed from the world model, its border will still appear on the right hand side, though it will be empty (rather than colored as it was on the left hand side).
- when an object is added to the world model, its border will be bold.
- when the robot is moved, it may be either translated or rotated, or both. This is emphasized by showing both the old location of the robot as an empty border (the same representation that is used for an object that has been removed), and as a colored object with a bold border (the same representation as for an object that has been added).

An example of the use of the rule editor is shown in Figure 5. In this example, the rule is a response to a sonar return from an unexpected object; the response is to insert an obstacle at the location returned by the sonar sensor.

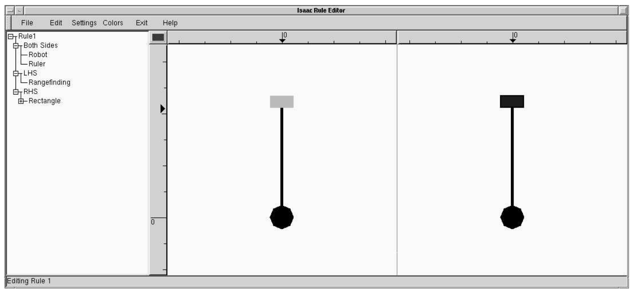


Figure 5. Rule editor

5. Device Interaction Library

Isaac is being developed for use in POSIX−compliant environments, in particular Linux with the Kansas University Real−time Extensions[8]. A library is being developed to provide a consistent interface to all of the sensors and actuators supported by Isaac.

Each sensor driver implements a method called `GetValue()`, which performs any interactions necessary with the operating system's device driver to obtain a reading from the sensor, and translates the reading into a 32 bit signed binary value.

Similarly, each actuator driver implements a function called `PutValue()`, which translates a 32 bit binary value into a value expected by the low−level device driver, and writes it to the driver.

All sensors and actuators of a common type accessed as elements of an array; the sensor and actuator drivers are loadable at run−time. Both sensors and actuators are required to provide an _init() method, which will open the device and perform any other initialization necessary to make use of it.

The intent of this mechanism is to provide as thin as possible an interface between the operating system and Isaac, providing the programmer with maximum flexibility in developing I/O objects.

6. Concluding Remarks

We have developed a powerful mechanism for describing input, output, and processing for a rule−based visual language in a consistent manner.

Our work to date has focussed on reactive robot control. While very successful for immediate response to inputs and for local navigation, it is not clear to what extent this processing model is appropriate for higher−level activities such as path planning. As development of Isaac progresses, it will be interesting to determine the extent to which our rule−based processing paradigm is appropriate for the construction of plans and more complex navigation than we have considered to date.

We have not considered path−planning at this time. However, a likely approach to the problem would be to allow a path−planning algorithm to insert new path objects in the world model. Alternatively, as Isaac is intended for geometric reasoning, it is likely that it will be a useful language for expressing path−planning algorithms which would run concurrently with the robot's operation.

REFERENCES

[1] Ambler, A.L., T. Green, T.D. Kimura, A. Repenning, and T.J. Smedley, ‘‘1997 Visual Programming Challenge Summary,’’ in *Proceedings of the 1997 IEEE Symposium on Visual Languages* (1997), 11−18.

[2] Brooks, R.A., ‘‘A robust layered control system for a mobile robot,’’ in *IEEE Journal of Robotics and Automation RA−2* (1986), 14−23.

[3] Cox, P. and T. Smedley, ‘‘Visual programming for robot control,’’ in *Proceedings of the 1998 IEEE Symposium on Visual Languages* (1998), 217−224.

[4] Gat, E.R., R. Desai, R. Ivleve, J. Loch, and D.P. Miller, ‘‘Behavior Control for robotic exploration of planetary surfaces,’’ in *IEEE Transactions on Robotics and Automation 10(4)* (1994), 490−503.

[5] Gindling, J., A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning, ‘‘LEGOsheets: a rule−based programming, simulation and manipulation environment for the LEGO programmable brick,’’ in *Proceedings of the 11th IEEE Symposium on Visual Langauges* (1995), 172−179.

[6] Harmon, S.Y., ‘‘A rule−based command language for a semi−autonomous Mars rover,’’ in *Mobile Robots IV*, W. J. Wolfe and W.H. Chun, eds (1989), 147−156.

[7] Heger, N., A. Cypher, and D. Smith, ‘‘Cocoa at the Visual Programming Challenge 1997,’’ in *Journal of Visual Languages & Computing 9* (1998), 151−168.

[8] Hill, R., B. Srinivasan, S. Pather, and D. Niehaus, *Temporal Resolution and Real−Time Extensions to Linux*, Technical Report ITTC−FY98−11510−03, Information and Telecommunication Technology Center, Department of Electrical Engineering and Computer Science, University of Kansas (1998).

[9] Meyrowitz, A.L., ‘‘Autonomous vehicles,’’ in *Proceedings of the IEEE 84(8)* (1996), 1147−1163.

[10] Meystel., A., *Autonomous Mobile Robots − Vehicles With Cognitive Control*, World Scientific Publishing Co. (1991).

[11] Orchard, R., *FuzzyCLIPS Version 6.04A User's Guide* (1998).

[12] Pfeiffer, J., ‘‘Altaira: a rule−based visual language for small mobile robots,’’ in *Journal of Visual Languages & Computing 9* (1998), 127−150.

[13] Pfeiffer, J.., ‘‘A language for geometric reasoning in mobile robots,’’ in *Proceedings of the IEEE Symposium on Visual Languages* (Tokyo Japan, September 1999), 164−171.

[14] Smith, D., A. Cypher, and J. Spohrer. ‘‘KidSim: programming agents without a programming language.’’ in C*ommunications of the ACM 37* (1994) 54−67.

[15] Vinyard, R., J. Pfeiffer, and B. Margolis, ‘‘Hardware Abstraction in a Visual Programming Environment,’’ submitted for publication.