How to Develop a RoboCupRescue Agent

for RoboCupRescue Simulation System version 0
1st edition

written by Takeshi Morimoto
edited by RoboCupRescue Technical Committee

mailing list: r-resc@isi.edu
web page: http://robomec.cs.kobe-u.ac.jp/robocup-rescue

Contents

List of Figures

List of Tables

1 Introduction

This manual is written for computer science/engineering researchers and students to acquire sufficient knowledge on how to develop a *RoboCupRescue agent (RCR agent)* without background knowledge on RoboCupRescue[1].

In the RoboCupRescue simulation, rescue agents such as ambulance teams and fire brigades act in large urban disasters. Soon after a large earthquake, buildings collapse, many civilians are buried in the collapsed buildings, fires are spreading, and it becomes difficult for rescue teams to pass roads because these are blocked by debris of buildings and something else. The objective of rescue agents is, coherently, to minimize damage resulting from disasters.

The urban area called the *disaster space* is simulated by the *RoboCupRescue Simulation System (RCRSS)*. The RCRSS consists of several modules, and the *kernel* module manages the whole of the RCRSS. An RCR agent also is one of modules, and communicates with the kernel through a network to act in the disaster space. Necessary for developing an RCR agent is understanding of the RCRSS, the disaster space, and the means by which an RCR agent communicates with the kernel.

This manual, at first, describes the RCRSS and the disaster space. Then it describes how an RCR agent acts in the disaster space by communicating with the kernel. Additionally, it shows how to use the RCRSS. The appendix contains an explanation of how to develop an RCR agent using the *YabAPI*, a bare essential API to develop an RCR agent in JAVA. The reader will be able to develop an original RCR agent soon, using YabAPI.

In this manual, the RCRSS means the RCRSS version 0.xx unless otherwise noted.

2 RoboCupRescue Simulation System

2.1 Structure

The RCRSS is a real-time distributed simulation system that is built of several modules connected through a network (Figure 1). Each module can run on different computers as an independent program, so the computational load of the simulation can be distributed to several computers. Each disaster phenomenon such as collapse of buildings and fire spread is simulated by a dedicated *sub-simulator* of each disaster. Ambulance teams and fire brigades act as several independent RCR agents. The geographical information system (*GIS*) provides initial condition of the disaster space, and the *viewer* visualizes conditions of the disaster space. The kernel manages communications among the modules and the simulation.

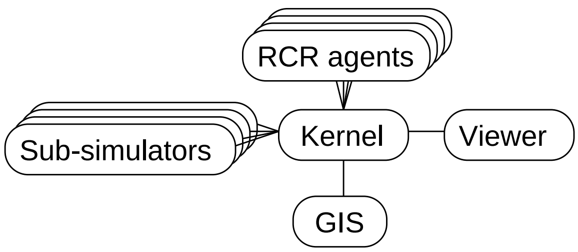


Figure 1: Structure of the RCRSS

2.2 Initialization and Progress

Before starting a simulation, the kernel integrates all modules into the RCRSS as follows.

1. The kernel connects to the GIS, and the GIS provides the kernel with initial condition of the disaster space.
2. Sub-simulators and the viewer connect to the kernel, and the kernel sends them the initial condition.
3. RCR agents connect to the kernel with their agent type. The kernel assigns each RCR agent to a rescue team and civilians etc. in the disaster space, and sends initial condition within each agent's cognition.

When all rescue teams and civilians in the disaster space have been assigned to RCR agents, the kernel finishes the integration and the initialization of the RCRSS. Then, the simulation starts. All sub-simulators and the viewer have to be connected to the kernel before all assignment of an RCR agent have been finished.

The simulation proceeds by repeating the following *cycle*. At the first cycle of the simulation, steps 1 and 2 are skipped.

1. The kernel sends individual vision information to each RCR agent.
2. Each RCR agent submits an action command to the kernel individually.
3. The kernel sends action commands of RCR agents to all sub-simulators.
4. Sub-simulators submit updated states of the disaster space to the kernel.
5. The kernel integrates the received states, and sends it to the viewer.
6. The kernel advances the simulation clock of the disaster space.

One cycle in the simulation corresponds to one minute in the disaster space. The kernel waits half a second for command/state submissions at steps 2 and 4, so it takes one second to simulate one cycle. It occasionally takes a few seconds according to the scale of simulation and machine specs. All RCR agents must decide an action within half a second.

The modules of the RCRSS work as follows at the beginning of the simulation.

1st cycle: A collapse sub-simulator simulates building collapse, and a fire sub-simulator starts simulating fire spread.

2nd cycle: A blockade sub-simulator simulates road blockade based on the result of the collapse simulator, and a *misc* (stands for miscellaneous) sub-simulator starts simulating humans who are buried and injured.

3rd cycle: RCR agents start acting.

2.3 Disaster Space

The kernel models the disaster space as a collection of *objects* such as buildings, roads, and humans (Figure 2). Each object has *properties* such as its position and shape, and is identified by a unique *ID*. The data type of a property is either a 32-bit signed integer or an integer array. Figure 2 illustrates objects and properties necessary and sufficient for developing RCR agents. For more information, refer to the manual of the RCRSS[2]. Immovable objects are located at the position designated by their

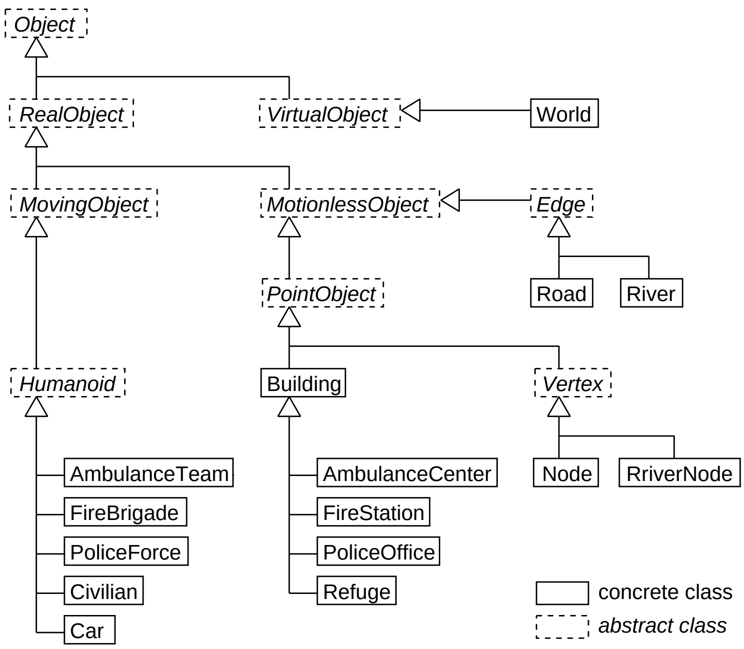
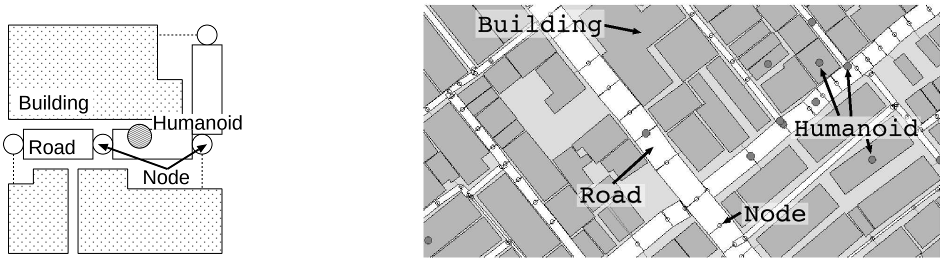


Figure 2: Class hierarchy of objects in the disaster space

geographical properties, and all objects are linked by their topological properties (Figure 3). A *Road* object models a road as a rectangle area (Table 2, Figure 5). A *Node* object models an entrance of a building or an end-point of a Road object (Table 3, Figure 6). The *Humanoid* object models a rescue team or a civilian family (Table 4, Figure 7).



(a) All objects are linked by their topological properties

(b) View of a disaster space

Figure 3: Disaster space

Table 1: Properties of a Building object

<table>
<tr><th>Property</th><th>Type or [Unit]</th><th>Comment</th></tr>
<tr><td>x, y</td><td>[mm]</td><td>The x-y coordinate of the representative point</td></tr>
<tr><td>entrances</td><td>$\mathrm{ID}_1, \cdots, \mathrm{ID}_n, 0$</td><td>The connected nodes and roads.</td></tr>
<tr><td>floors</td><td>[floor]</td><td>The number of floors</td></tr>
<tr><td>buildingAreaGround</td><td>[mm^2]</td><td>The area of the ground floor (1st floor)</td></tr>
<tr><td>buildingAreaTotal</td><td>[mm^2]</td><td>The total area summing up all floors</td></tr>
<tr><td>fieryness</td><td colspan="2">The state that specifies how much it is burning
<table>
<tr><td>0</td><td>Unburned</td><td></td></tr>
<tr><td>
1
2
3</td><td>Burning</td><td>Burning time rate
0.00 ∼ 0.33
0.33 ∼ 0.67
0.67 ∼ 1.00</td></tr>
<tr><td>
5
6
7</td><td>Put out</td><td>Burned rate
0.0 ∼ 0.2
0.2 ∼ 0.7
0.7 ∼ 1.0</td></tr>
</table></td></tr>
<tr><td>buildingCode</td><td colspan="2">The code of a construction method
<table>
<tr><th>Code</th><th>Construction method</th><th>Fire transmission rate</th></tr>
<tr><td>0</td><td>Wooden</td><td>1.8</td></tr>
<tr><td>1</td><td>Steel frame</td><td>1.8</td></tr>
<tr><td>2</td><td>Reinforced concrete</td><td>1.0</td></tr>
</table></td></tr>
</table>

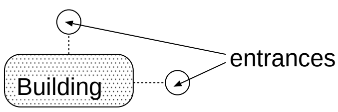


Figure 4: Building object

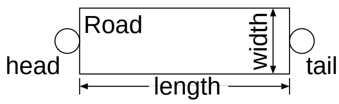


Figure 5: Road object

Table 2: Properties of a Road object

Property	Type or [Unit]	Comment
head, tail	ID	The end-point. It must be a node or a building
length, width	[mm]	The length from the head to the tail, and the width
linesToHead/Tail	[line]	The number of traffic lanes for cars toward the head/tail
block	[mm]	The width of a blocked part through which cars cannot pass
repairCost	[team·cycle]	The cost required for clearing the block

Table 3: Properties of a Node object

Property	Type or [Unit]	Comment
x, y	[mm]	The x-y coordinate
edges	$\mathrm{ID}_1, \cdots, \mathrm{ID}_n, 0$	The connected roads and buildings

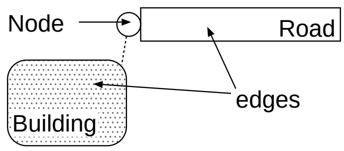


Figure 6: Node object

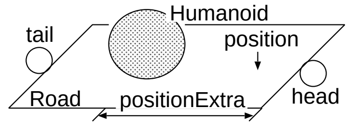


Figure 7: Humanoid object

Table 4: Properties of a Humanoid object

Property	Type or [Unit]	Comment
position	ID	An object that the humanoid is on. When the humanoid is loaded by an ambulance, this is set to the ambulance.
positionExtra	[mm]	The offset length from the position: a length from the head when the humanoid is on a road, otherwise it is zero
hp	[health point]	The health point. The humanoid dies when this becomes zero
damage	[health point]	The damage point by which the hp dwindles every cycle. This becomes zero immediately after the humanoid arrives at a refuge
buriedness	[team·cycle]	The cost required to rescue the humanoid. When this is greater than zero, the humanoid cannot act.

Every object has a representative point, and the distance between two objects is calculated from their representative points. A representative point of a Building and a Node object is a point plotted by the x and y properties. A Road object's is a midpoint between its head and tail in this regard a fraction is rounded down. If a Humanoid object is on a Building, Node, or AmbulanceTeam object, the representative point is the same as that of the object under the Humanoid object. If a Humanoid object is on a Road object, the representative point is a point positionExtra [mm] away from the head of the Road object.

3 RoboCupRescue Agent

3.1 Definition

An RCR agent controls act of an object in the disaster space. The object is called a *controlled object*, and there are seven classes for it: the Civilian, AmbulanceTeam, FireBrigade, PoliceForce, AmbulanceCenter, FireStation, and PoliceOffice. An RCR agent controlling act of a Civilian object is called a *civilian agent*, an RCR agent controlling act of an AmbulanceTeam object is called an *ambulance team agent*, and so on. Additionally, the ambulance team, fire brigade, and police force agent are collectively called a *platoon agent*, and the ambulance center, fire station, police office agent are also called a *center agent*. The platoon and center agent is called a *rescue agent*.

Act of an object is processed as repeating cognition of the surrounding circumstances and decision of an act at each cycle. An RCR agent recognizes the surrounding circumstances based upon vision information received from the kernel, decides an act, and submits the act to the kernel. Moreover, an RCR agent communicates with other RCR agents asynchronously (i.e. independently of cycles).

An RCR agent has different capabilities for cognition and act according to its type (Table 5). An RCR agent gets vision information by the *sense* capability and auditory information by the *hear* capability, and acts by the *move*, *rescue*, *load*, *unload*, *extinguish*, and *clear* capabilities, and utters natural voice by *say* capability and speak via telecommunication by *tell* capability.

Table 5: Capabilities of RCR agents

Type	Capabilities
Civilian	Sense, Hear, Say, Move
Ambulance Team	Sense, Hear, Say, Tell, Move, Rescue, Load, Unload
Fire Brigade	Sense, Hear, Say, Tell, Move, Extinguish
Police Force	Sense, Hear, Say, Tell, Move, Clear
Ambulance Center	Sense, Hear, Say, Tell
Fire Station	Sense, Hear, Say, Tell
Police Office	Sense, Hear, Say, Tell

3.2 Protocol of Communication with the Kernel

3.2.1 RCRSS Protocol

An RCR agent communicates with the kernel through a network using the *RCRSS protocol*. A data unit for the RCRSS protocol is called a *block* which consists of a *header*, *body length*, and *body* field (Figure 8), and a packet of the RCRSS protocol consists of zero or more blocks and a *HEADER_NULL (0x00)* as a terminator (Figure 9). The format of the body field depends upon the header. The body length field is set the byte size of the body.

0 31

header
body length
body ⋮

Figure 8: Block of the RCRSS protocol

block_1	block_2	⋯	block_n	HEADER_NULL

Figure 9: RCRSS protocol packet

Table 6 shows headers related to RCR agents. An RCRSS protocol block having some header H is called an H block, and a body of an H block is called an H body for short. Moreover, a block issued to submit the will to act such as an AK_MOVE and an AK_REST block is called an *action command*. A block for communication such as an AK_SAY block is called a *communication command*.

Table 6: Header and its use

Value	Header	Use
	To the kernel:	
0x10	AK_CONNECT	To request for the connection to the kernel
0x11	AK_ACKNOWLEDGE	To acknowledge for the KA_CONNECT_OK
0x81	AK_MOVE	To submit the will to move to another position
0x88	AK_RESCUE	To submit the will to rescue an humanoid
0x82	AK_LOAD	To submit the will to load an humanoid
0x83	AK_UNLOAD	To submit the will to unload an humanoid
0x86	AK_EXTINGUISH	To submit the will to extinguish a fire
0x89	AK_CLEAR	To submit the will to clear a blockade
0x80	AK_REST	To submit the will to do nothing
0x84	AK_SAY	To submit the will to say something
0x85	AK_TELL	To submit the will to tell something
	From the kernel:	
0x50	KA_CONNECT_OK	To inform of the success of the connection
0x51	KA_CONNECT_ERROR	To inform of the failure of the connection
0x52	KA_SENSE	To send vision information
0x53	KA_HEAR	To send auditory information

The body field consists of 32-bit integers such as a time and an ID, strings, and objects serialized into binary data. Body field formats will be defined in §3.4. Here we describe the data types and structures of the elements constructing a body field.

int An *int* element is a 32-bit signed integer. It represents x coordinate, an ID for the position property, etc.

IDs An *IDs* element consists of zero or more IDs and 0 as a terminator (Figure 10). It represents the entrances, edges, etc.

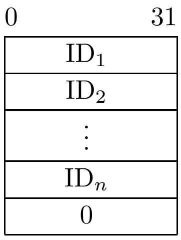


Figure 10: IDs element

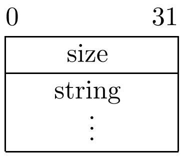


Figure 11: String element

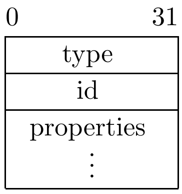


Figure 12: Object element

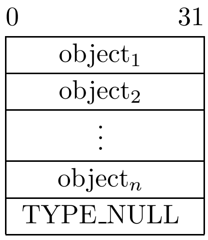


Figure 13: Objects element

String A *String* element consists of a byte size of the string and an ASCII character string (Figure 11). The string is aligned to a multiple of 4 bytes, but the byte size is set the original size. This element is used for inter-agent communications etc.

Object An *Object* element represents an object in the disaster space, and consists of the type, ID, and properties of the object (Figure 12). The type is either one listed in Table 7, and the properties consist of zero or more properties $< type, value >$ and a *PROPERTY_NULL (0x00)* as a terminator:

$$\{< type_1, value_1 >, \cdots, < type_n, value_n >, \ \textit{PROPERTY\_NULL}\}.$$

The type of a property is either one listed in Table 8. A value for properties assigned to 0x00 $\sim$ 0x7F is an int element, and for properties assigned to 0xC0 $\sim$ 0xFF is an IDs element. For other property types, refer to the manual of the RCRSS[2]. An Object element is used for the kernel to send vision information to an RCR agent.

Table 7: Value of object types

Type	Value
Civilian	0xE8
Fire brigade	0xE9
Ambulance team	0xEA
Police force	0xEB
Road	0xA8
Node	0xC8
Building	0xB0
Refuge	0xB8
Fire station	0xB9
Ambulance center	0xBA
Police office	0xBB
River	0xA9
RiverNode	0xC9

Table 8: Value of property types

Property	Value
head	0x0C
tail	0x0D
length	0x18
width	0x26
linesToHead	0x29
linesToTail	0x2A
block	0x16
repairCost	0x27
x	0x03
y	0x04
edges	0xF2
entrances	0xEB
floors	0x0E
buildingAreaGround	0x33
buildingAreaTotal	0x34
fieryness	0x10
buildingCode	0x32
position	0x06
positionExtra	0x07
hp	0x0A
damage	0x0B
buriedness	0x17

Objects An *Objects* element consists of zero or more Object elements and a terminator *TYPE_NULL (0x00)* (Figure 13). The kernel sends information of objects in the disaster space to an RCR agent by the Objects element: however, it does not necessarily contain all objects and properties; that is, the kernel sends only differences from those that it sent last time.

3.2.2 LongUDP

The RCRSS protocol assumes that the *LongUDP*. The LongUDP provides a procedure for application programs to send large size data to other programs with a simple protocol mechanism using UDP (User Datagram Protocol) as the underlying protocol. In the mechanism, the sender program divides data into several UDP packets with adding a *LongUDP header* (Figure 14), and the receiver program rebuilds data from the UDP packets sent from the same sender and assigned the same LongUDP ID.

0 … 15	16 … 31
0x008	id
number	total

0x008 ··· The magic number
id ··· An ID of the LongUDP packet
number ··· An ordinal number of the UDP packet in the *total* ($0 \le number < total$)
total ··· A number of total UDP packets building the LongUDP packet

Figure 14: LongUDP header format

Finally, the relation among the RCRSS protocol, LongUDP, and UDP is summarized in Figure 15.

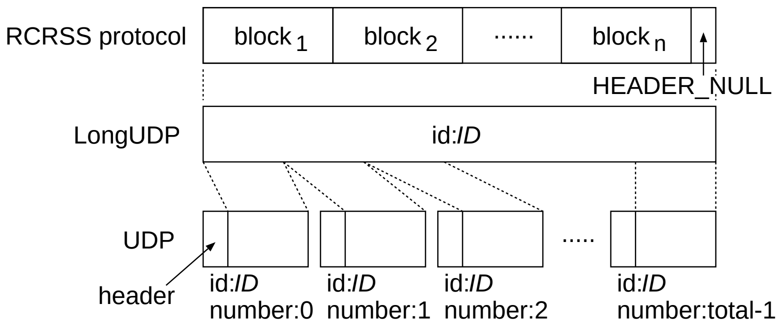


Figure 15: RCRSS Protocol Stack

3.3 Process Flow

Before starting simulation, every RCR agent has to be assigned to its controlling object by communicating with the kernel (Figure16). Each RCR agent, at first, requests the kernel for the connection by submitting an AK_CONNECT block to the listening port 6000 of the host where the kernel is running. Then, the kernel replies to the RCR agent with a KA_CONNECT_OK block when the connection is established, or sends a KA_CONNECT_ERROR to tell failure. In either case the kernel sends the block from another port than the listening port, and the RCR agent communicates with the kernel through the sources of the AK_CONNECT and the KA_CONNECT_OK block after this step (Table 9). The KA_CONNECT_OK block contains the geographical information of the whole of the disaster space and vision information, and the RCR agent initializes itself based upon the information. Finally, the RCR agent acknowledges the KA_CONNECT_OK block with an AK_ACKNOWLEDGE block.

At every cycle in the simulation, each RCR agent receives a KA_SENSE block as its own vision information from the kernel, and then submits an action command to the kernel (Figure 17). An RCR agent, moreover, can communicate with other RCR agents by communication commands asynchronously of the cycle. The kernel sends KA_HEAR blocks as auditory information to receiving RCR agents soon after reception of a communication command (Figure 18).

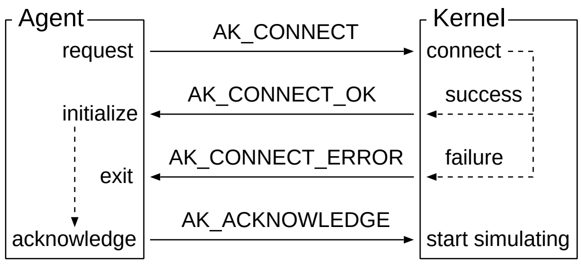


Figure 16: Initializing Process

Table 9: Source and Destination of RCRSS protocol blocks

Packet	Source	Destination
AK_CONNECT	Arbitrary*^{a}	Listening Port of the Kernel*^{b}
KA_CONNECT_OK	Arbitrary	Source of *AKConn**^{c}
KA_CONNECT_ERROR	Arbitrary	Source of *AKConn*
AK_ACKNOWLEDGE	Source of *AKConn*	Source of *KAConnOk**^{d}
KA_SENSE	Arbitrary	Source of *AKConn*
action	Source of *AKConn*	Source of *KAConnOk*
KA_HEAR	Arbitrary	Source of *AKConn*
communication	Source of *AKConn*	Source of *KAConnOk*

*a A source of *AKConn* may be shared with more than one RCR agents.
*b It is specified as port 6000 of the host where the kernel is running.
*c *AKConn* stands for the AK_CONNECT block.
*d *KAConnOk* stands for the KA_CONNECT_OK block.

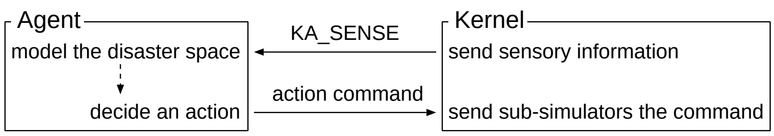


Figure 17: Perception and action at each cycle

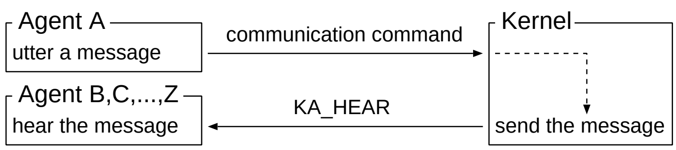


Figure 18: Inter-agent communication

3.4 Initialization

Here we describe the body of RCRSS protocol blocks used to initialize an RCR agent.

AK_CONNECT body

An AK_CONNECT block is used for an RCR agent to request for the connection to the kernel. Its body consists of three int elements: temporaryId, version, and agentType (Figure 19). The temporaryId field is used to identify the RCR agent submitting an AK_CONNECT block from the socket from which other RCR agents may also submit a block. Although the version must be 0 in the RCRSS manual[2], informally, the version may be 1. In this case, a map field of a KA_CONNECT_OK is not sent in order for many RCR agents to establish the connection as fast as possible.

Table 10: Value of agent types

Type	Value
Civilian	1
Fire brigade	2
Fire station	4
Ambulance team	8
Ambulance center	16
Police force	32
Police office	64

0 31	
int temporaryId	An arbitrary ID
int version	This must be 0
int agentType	The type of the agent. This is either one listed in Table 10

Figure 19: AK_CONNECT body

KA_CONNECT_OK body

A KA_CONNECT_OK block is sent from the kernel, and the RCR agent can get information of its controlling object and the whole of the disaster space. Its body consists of temporaryId, id, self, and map (Figure 20).

0 31	
int temporaryId	The same value as that of the AK_CONNECT block
int id	The ID of the *self*
Object self ⋮	The object which the RCR agent controls
Objects map ⋮	The information about the whole of the disaster space

Figure 20: KA_CONNECT_OK body

KA_CONNECT_ERROR body

A KA_CONNECT_ERROR block is sent from the kernel in order to tell failure of the connection. Its body consists of the temporaryId and the reason (Figure 21). There are two reasons mainly:

- "unknown version" means that the given version by the AK_CONNECT block is invalid, and
- "no more agent" means that all controlled objects whose type is the same as the given agentType by the AK_CONNECT block have already been assigned to other RCR agents.

0 31	
int temporaryId	The same value as the AK_CONNECT block's
String reason ⋮	The reason why the connection failed

Figure 21: KA_CONNECT_ERROR body

AK_ACKNOWLEDGE body

An AK_ACKNOWLEDGE block is used for the RCR agent to acknowledge the KA_CONNECT_OK block. Its body consists of the ID of its controlling object (Figure 22).

0 31	
int id	The ID of its controlling object

Figure 22: AK_ACKNOWLEDGE body

3.5 World Modeling

An RCR agent has to construct a model that represents the disaster space in its interior based upon vision information contained in a KA_SENSE block at each cycle. A KA_SENSE body consists of id, time, self, and map (Figure 23). The time may be regarded as the number of cycles from the start of the simulation. The self and the map contain only differences from the last cycle. The map contains information of objects within a radius of 10 m, and all fires. Note that a distance between two objects is calculated from their representative points (See §2.3). For instance, in Figure 24, an RCR agent controlling the self can get information of the Node1 and the Road2, but cannot get information of the Road1 where the self is (!) and the Building1. In addition, even if the self passed a route last cycle, the RCR agent may not be able to get information of objects near the passed route. That is to say, an RCR agent can get only a snapshot at the beginning of each cycle.

0 31	
int id	The ID of the *self*
int time	Current time in the disaster space
Object self ⋮	The object which the RCR agent controls
Objects map ⋮	The vision information

Figure 23: KA_SENSE body

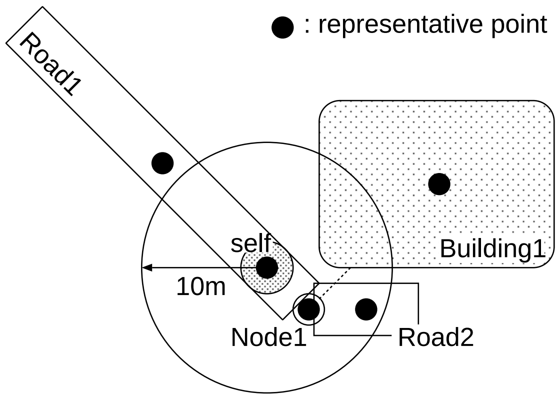


Figure 24: Visual range

3.6 Action

After receiving a KA_SENSE block, an RCR agent submits an action command as the will of its controlling object at each cycle. Although an RCR agent may submit two or more action commands at one cycle, the kernel adopts only the last one. As mentioned in §2.2, an RCR agent needs to submit an action command before taking half second after receiving a KA_SENSE block, or the kernel regards that the RCR agent dose not want to act at the cycle. Moreover, delayed action commands are ignored. Note that a buried humanoid, whose buriedness is greater than zero, cannot act.

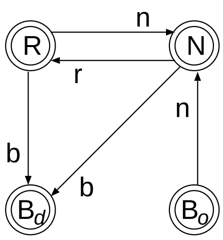
3.6.1 Move

A humanoid can move in the disaster space — a platoon can drive a car and a civilian family can walk — by submitting an AK_MOVE block (Figure 25). The routePlan must be a statement acceptable by an automaton shown in Figure 26, which consists of the current position as the origin and a series of objects reaching the destination. When the humanoid is loaded by an ambulance, the origin is the ambulance's position. Every object that comprises a routePlan must be a MotionlessObject.

0 31	
int selfId	Its controlling object's ID
IDs routePlan ⋮	The route plan along which the agent want to move

Figure 25: AK_MOVE body

A humanoid can move only if it specifies the route plan, because the traffic sub-simulator has to simulate the move of all humanoids as they requested as possible in a limit time. But a humanoid can pass neither a blockade nor a traffic jam, so a humanoid agent has to specify a route plan avoiding them. Here we describe blockade, traffic jam, and the maximum speed of moving objects. For more information, refer to the specification for the traffic sub-simulator contained as the README.txt file in [3].



Initial state	Current position
R	Road
N	Node
B_o	Building

Input	Meaning
r	Adjacent road ID
n	Adjacent node ID
b	Adjacent building ID

Figure 26: Automaton that accepts route plans

Blockade A blockade is assumed to be located at the midpoint of a Road object. On a road where there are more than one traffic lane, the lanes are blocked from outside to inside. When a humanoid is on a road, the definition of a blockade is as follows, where every variable is a property of the road except for a positionExtra property of the humanoid.

$$
\begin{aligned}
&lineWidth := \frac{width}{linesToHead + linesToTail} \\
&blockedLines := \left\lfloor \frac{block}{2 \cdot lineWidth} + 0.5 \right\rfloor \\
&passableLinesToHead := max(0,\ linesToHead - blockedLines) \\
&passableLinesToTail\ \ := max(0,\ linesToTail\ \ - blockedLines) \\
&passableLinesToHead = 0,\ positionExtra > \frac{length}{2}\ \rightarrow\ isBlockedToHead \\
&passableLinesToTail\ \ = 0,\ positionExtra < \frac{length}{2}\ \rightarrow\ isBlockedToTail
\end{aligned}
$$

If the *isBlockedToHead/Tail* is true, the humanoid cannot pass the road toward its head/tail. If false while there is a blockade on the road, it means that several inside lanes are not blocked or at least the humanoid has already bypassed the blockade.

Traffic Jam A traffic jam may occur on roads where many humanoids concentrate, because each humanoid moves keeping a safe distance to its forward moving object. The minimum safe distance to a forward platoon from a humanoid is 8 m, and to a forward civilian from a civilian is 1 m. Cars for rescue are, however, prioritized in the traffic, so that platoons can ignore civilians even if they are on the way, assuming that civilians are attentive to cars and give way to them. If there is a stopped humanoid on the way, another humanoid who wants pass the way changes to a lane of the same direction, and then can pass.

Maximum speed The maximum speed of a platoon is 20 [km/h], so a platoon can move 333 m in one cycle at most. The civilian's maximum speed is 3 [km/h]. Actually, a humanoid seldom moves at the maximum speed through one cycle, because a humanoid has to pass cross-points, change lanes, avoid blockades, etc.

3.6.2 Rescue

An ambulance team can progressively rescue buried humanoids under collapsed buildings by submitting an AK_RESCUE block (Figure 27). Rescuing a humanoid by an ambulance team in a cycle reduces the buriedness of the humanoid by 1 [team·cycle]. If more ambulance teams work on rescuing a humanoid, the humanoid can be rescued in less cycles. The target humanoid must be at the same position as the ambulance team. Note, however, that because a road and its end-point nodes are considered as the same position, the ambulance team can rescue a buried humanoid midst on a road from an end-point node of the road.

0 31	
int selfId	Its controlling object ID
int targetId	The humanoid ID the self rescues

Figure 27: AK_RESCUE body

3.6.3 Carry of the Injured

An ambulance team can load a humanoid to its car by submitting an AK_LOAD block (Figure 28), and can unload a humanoid from its car by submitting AK_UNLOAD block (Figure 29). The target humanoid of an AK_LOAD block must be at the same position as the ambulance team similarly to a rescue command, and moreover, a buriedness of the humanoid must be zero. Note that an ambulance team can neither load nor unload a moving humanoid even if the position of the humanoid does not change by moving, because AK_MOVE blocks are dealt with before AK_LOAD/AK_UNLOAD blocks. So an injured humanoid must not move in order to be carried to a refuge. After unloading, the injured humanoid will stand initially at the same position as the ambulance team. If an ambulance team agent submits an AK_UNLOAD block before loading, the ambulance team does nothing at the cycle.

0 31	
int selfId	Its controlling object ID
int targetId	The humanoid ID, which the self loads

Figure 28: AK_LOAD body

0 31	
int selfId	Its controlling object ID

Figure 29: AK_UNLOAD body

3.6.4 Extinguishing a Fire

A fire brigade can extinguish a fire by submitting an AK_EXTINGUISH block (Figure 30) which contains fire hose's nozzle elements (Figure 31), and a fire brigade may use more than one fire hoses for different fires simultaneously. The direction of a nozzle element should be designated as zero degree for Y-axis positive direction, and up to 1295999 in the unit of second in the counterclockwise direction. Although the effect of extinguishing is not defined explicitly, it may not be difficult for even a few fire brigades to extinguish an early fire. On the contrary, it is difficult for even many to extinguish a late and big fire.

0 31
int selfId
Nozzle nzl_1 ⋮
⋮
Nozzle nzl_n ⋮
int 0

Figure 30: AK_EXTINGUISH body

0 31	
int targetId	The ID of a building for which the nozzle headed
int direction	The direction from the nozzle to the target
int x	The x coordinate of the nozzle
int y	The y coordinate of the nozzle
int quantity	The water quantity discharged from the nozzle in a cycle. A unit is 0.001 [m^3]

Figure 31: Nozzle element

3.6.5 Clearing a Blockade

A police force can progressively clear debris on a road by submitting an AK_CLEAR block (Figure 32). Clearing debris on a road by a police force in a cycle reduces a block of the road by 1/repairCost %, and a repairCost of the road by 1 [team·cycle]. If more police forces work on clearing debris on a road, the road can become passable in less cycles. The target road must be at the same position as the police force similarly to a rescue command.

0 31	
int selfId	Its controlling object ID
int targetId	The road ID, which the self clears

Figure 32: AK_CLEAR body

3.6.6 Doing Nothing

When a controlled object does nothing at a cycle, the RCR agent may submit a AK_REST block (Figure 33). The block is used, for example, to cancel previous commands submitted at the cycle.

0 31	
int selfId	Its controlling object ID

Figure 33: AK_REST body

3.7 Inter-agent Communication

An RCR agent can communicate with other RCR agents; an RCR agent can utter natural voice by submitting AK_SAY block, and a platoon and a center, furthermore, can use telecommunication by submitting AK_TELL block (Figure 34); and an RCR agent can hear these messages by receiving KA_HEAR blocks (Figure 35). Natural voice can be heard by humanoids within a radius of 30 m, and telecommunication can be heard by platoons and the center of the same type as the speaker. Furthermore, telecommunication by a center is transferred to other type centers, too (Figure 36). In each case, auditory information is sent soon after a communication command is submitted.

0 31	
int selfId	Its controlling object ID
String message	The message self utters

Figure 34: AK_SAY/AK_TELL body

0 31	
int selfId	Its controlling object ID
int senderId	The sender object ID
String message ⋮	The message uttered by the sender

Figure 35: KA_HEAR body

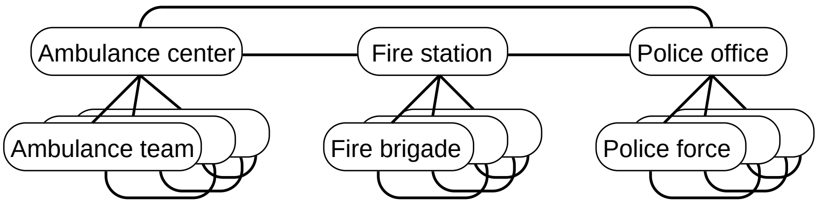


Figure 36: Telecommunication network

For reference, here we show relations between a command and a module handling the command in the RCRSS version 0.40 (Table 11).

Table 11: Module's handling commands

Module	Handling commands
Kernel	AK_SAY, AK_TELL
Traffic sub-simulator	AK_MOVE, AK_LOAD, AK_UNLOAD
Fire sub-simulator	AK_EXTINGUISH
Misc sub-simulator*a	AK_RESCUE, AK_CLEAR

*a The misc sub-simulator simulates the hp, damage, buriedness, etc. properties.

4 How to Use the RCRSS

The RCRSS basic package[4] contains most of the necessaries for developing RCR agents, and the RCRSS can be installed and started by only a few commands (See the READ_ME.txt file in the package). In the RUN directory of the package, there are shell scripts and configuration files for the RCRSS. The all.sh file executes all modules except for rescue agent modules on a single PC, and the simulation can be started by connecting rescue agents to the kernel. The YabAI.sh file executes sample rescue agents. For the simulation of large area, it is necessary to execute the RCRSS with at least three PCs, each of which is responsible to:

- Rescue agents,
- Fire sub-simulator and civilian agents, and
- Kernel and other modules.

We have confirmed the simulation of the tenth part of the Kobe city map by using three PCs equipped with a Pentium III 930 MHz processor and 256 MB of main memory. In order to distribute a computational load of the simulation to several PCs, it is necessary to specify the name of a host where the kernel runs as an argument of each shell script executing a module, and to make modules run on several PCs. For example, the following commands execute the GIS and the kernel on PC1, and the sample civilian agents on PC2.

```
PC1 % cd RUN/
PC1 % ./0gis.sh &
PC1 % ./1kernel.sh

PC2 % cd RUN/
PC2 % ./samplecivilian.sh PC1
```

In addition, the RCRSS requires time to initialize itself in order to send a large amount of information of the disaster space.

The log of the simulation is stored into the rescue.log file which can be used to replay the simulation by executing the logviewer.sh file. However, actions of RCR agents are not stored. Viewers and debuggers such as [5, 6] store the actions, and therefore enables to debug RCR agents and sub-simulators.

Table 12 shows configuration files in the RUN directory. The config.txt file configures the behavior of the RCRSS, and can be edited by a text editor. Detailed information of the config.txt is written as comment in the parameters.hxx file in the kernel directory. The gisini.txt file configures how many controlled objects, ignition points, and refuges exist in the disaster space, and where these are. An ignition point is a building ignited by the fire sub-simulator at the first cycle. The gisini.txt file is made by the initial position setting tool available from [7]. The galpolydata.dat and the shindopolydata.dat files are used to simulate building collapse and road blockade, respectively, and are made by the earthquake distribution data creating tool[8]. The building.bin, road.bin, and node.bin files are geographic information data of buildings, roads, and nodes, respectively, in the disaster space, and available from [7]. The RCRSS can simulate various areas by switching geographic information data. For the present, the data of Kobe city which had been damaged seriously at the Hanshin-Awaji earthquake at 1995 in Japan, and some virtual city data, etc. are available. The package version 0.40 contains the one tenth part of the Kobe city data.

Table 12: Configuration files in the RUN directory

File	Object
config.txt	Behavior of the RCRSS
gisini.txt	All controlled objects, refuges, ignited buildings
galpolydata.dat	Distribution of ground acceleration
shindopolydata.dat	Distribution of seismic intensity
building.bin	Position, shape, etc. of buildings
road.bin	Position, shape, etc. of roads
node.bin	Position etc. of nodes

References

[1] *RoboCupRescue Official Site*,
`http://robomec.cs.kobe-u.ac.jp/robocup-rescue/`.

[2] Tomoichi Takahashi, *RoboCupRescue Simulator Manual*, 2000,
`http://kiyosu.isc.chubu.ac.jp/robocup/Rescue/`.

[3] Takeshi Morimoto, *Traffic Simulator for RoboCupRescue Simulation System*, 2002,
`http://ne.cs.uec.ac.jp/~morimoto/rescue/traffic/`.

[4] Tetsuhiko Koto, *RoboCupRescue Simulation System Basic Package*, 2002,
`http://ne.cs.uec.ac.jp/~koto/rescue/`.

[5] Takeshi Morimoto, *Viewer for RoboCupRescue Simulation System*, 2002,
`http://ne.cs.uec.ac.jp/~morimoto/rescue/viewer/`.

[6] Takeshi Morimoto, *Debugger for RoboCupRescue Simulation System*, 2002,
`http://ne.cs.uec.ac.jp/~morimoto/rescue/debugger/`.

[7] Michinori Hatayama, *GIS Component & Geographic Infomation Data for RoboCupRescue Simulation System*, 2002,
`http://www.cs.dis.titech.ac.jp/~hatayama/rcr/rcr_gis.html`.

[8] Tomoichi Takahashi, *Earthquake Distribution Data Creating Tool*, 2002,
`http://kiyosu.isc.chubu.ac.jp/robocup/Rescue/2002memo/GIS/tool.tar.gz`.

[9] Kosuke Shinoda, *RoboCupRescue Civilian Agent*, 2002,
`http://www.carc.aist.go.jp/~kshinoda/R_Civilian/`.

Appendix

A YabAPI — API to Develop an RCR Agent

The YabAPI is a bare essential API to develop an RCR agent in JAVA, which consists of four packages (Figure 37):

- the `yab.io` package provides functions for communication between an RCR agent and the kernel,
- the `yab.io.object` package provides classes of objects in the disaster space,
- the `yab.agent.object` package provides useful classes of objects in the disaster space for RCR agent developers. They wrap the `yab.io.object` package's classes, and
- the `yab.agent` package provides the skeletons of RCR agents and utilities for concisely describing their intelligence.

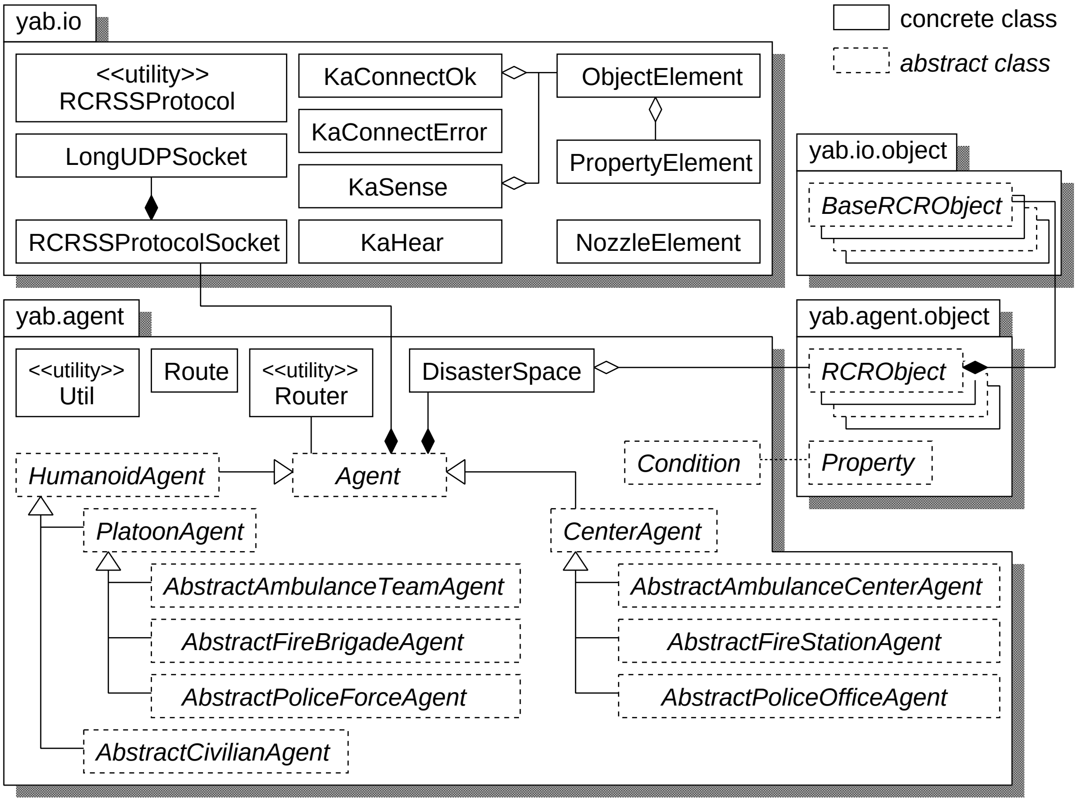


Figure 37: YabAPI

This appendix shows sample programs that use these packages. For more information, refer to the YabAPI's documentation in the yabapi/doc directory.

For C++ programmers, there also is a library *Agent Development Kit (ADK)* by Michael Bowling, which is available from `http://www-2.cs.cmu.edu/~mhb/research/rescue/`.

A.1 Communication with the kernel

The following program uses the `yab.io` package for an RCR agent to connect to the kernel and then control actions of its assigned object.

```
import java.net.*;
import yab.io.*;

class RCRAgent {
```

```
    RCRSSProtocolSocket socket;

    RCRAgent(int agentType, InetAddress kernelAddress, int kernelPort) {
        socket = new RCRSSProtocolSocket(kernelAddress, kernelPort);
        socket.akConnect(TEMPORARY_ID, VERSION, agentType);
        Object data = socket.receive();
        if (data instanceof KaConnectError)
            quit();
        KaConnectOk ok = (KaConnectOk)data;
        initialize(worldModel, ok);
        socket.akAcknowledge(ok.selfId);
    }

    void control() {
        while (true) {
            Object data = socket.receive();
            if (data instanceof KaSense) {
                update(worldModel, (KaSense)data);
                act();
            } else
                hear((KaHear)data);
        }
    }

    void act() {
        int[] routePlan = ...;
        socket.akMove(self.id, routePlan);
    }
}
```

A.2 World Modeling

The following program uses the `yab.io.object` and `yab.io` packages for an RCR agent to construct and update its interior world model. The `yab.io` package defines the same classes as Figure 2 in §2.3 except that the `Object` class is named `BaseRCRObject`.

```
import java.util.*;
import yab.io.object.*;
import yab.io.ObjectElement;

class DisasterSpace {
    HashMap idObjMap = new HashMap();
    BaseRealObject self;
    int time;

    DisasterSpace(int selfId, ObjectElement[] objs) {
        update(objs, INITIAL_TIME);
        self = get(selfId);
    }

    void add(BaseRealObject obj) { idObjMap.put(new Integer(obj.id), obj); }

    BaseRealObject get(int id) {
        return (BaseRealObject)idObjMap.get(new Integer(id));
    }
```

```
    void update(ObjectElement[] objs, int time) {
        this.time = time;
        for (int i = 0; i < objs.length; i ++)
            update(objs[i]);
    }

    void update(ObjectElement oe) {
        BaseRCRObject obj = get(oe.id);
        if (obj == null) {
            obj = BaseRCRObject.produce(oe.type, oe.id);
            if (obj instanceof BaseRealObject)
                add((BaseRealObject)obj);
        }
        obj.setProperty(oe.properties);
    }
}
```

A.3 Description of Intelligence

The `yab.agent` and `yab.agent.object` packages are used to concisely describe intelligence of RCR agents by providing:

- the skeletons of RCR agents,
- a method for getting a route plan, and
- the operators for a set of objects.

A.3.1 RCR agent Skeleton

The `yab.agent.AbstractXxxAgent` class, a skeleton of an RCR agent, is used to develop RCR agents. The developers have to only describe action of each cycle in `act` method and reaction to auditory information in `hear` method. For center agents, however, it is not necessary to implement `act` method.

```
import java.net.*;
import yab.agent.*;
import yab.agent.object.*;

class FireBrigadeAgent extends AbstractFireBrigadeAgent {
    FireBrigadeAgent(InetAddress address, int port) { super(address, port); }

    protected void act() throws ActionCommandException {
        // action of each cycle
    }

    protected void hear(RealObject sender, String message) {
        // reaction to auditory information
    }
}
```

`ActionCommandException` is thrown by methods that submit action commands such as `move` and `extinguish` methods, and is used to break out from `act` method. Hereby, action can be described like a rule set as the following.

```
    protected void act() throws ActionCommandException {
        extinguishNearFire();
        moveToFire();
```

```
    }

    protected void extinguishNearFire() throws ActionCommandException {
        if (there_are_fires_near_here)
            extinguish(one_of_near_fires);
    }

    protected void moveToFire() throws ActionCommandException {
        if (there_are_fires)
            move(route_to_one_of_fires);
    }
```

A.3.2 Route Planning

`Router.get` method takes as parameters an origin, destinations, and a function estimating moving cost, and returns the minimum cost route reaching one of the destinations from the origin.

```
    protected void move(Collection destinations) throws ActionCommandException {
        Route routePlan = Router.get(self.position(), destinations, cost_function);
        move(routePlan.toIDs());
    }
```

The `move` method of the humanoid agent skeleton takes only destinations parameter, and returns a route by estimating moving cost based upon passability of roads. For details about routing, refer to the documentation of `Router` class.

A.3.3 Set Operation

The `yab.agent.object.Property` and `yab.agent.Condition` classes are used to operate on a set of objects.

```
    Property fieryness = Property.get("Building", "fieryness");
    Condition isBurning = fieryness.gte(1).and(fieryness.lte(3));
    Collection fires = isBurning.extract(world.buildings);

    Property entrance = Property.get("Building", "entrance");
    Collection entrancesOfFires = entrance.collect(fires);
```