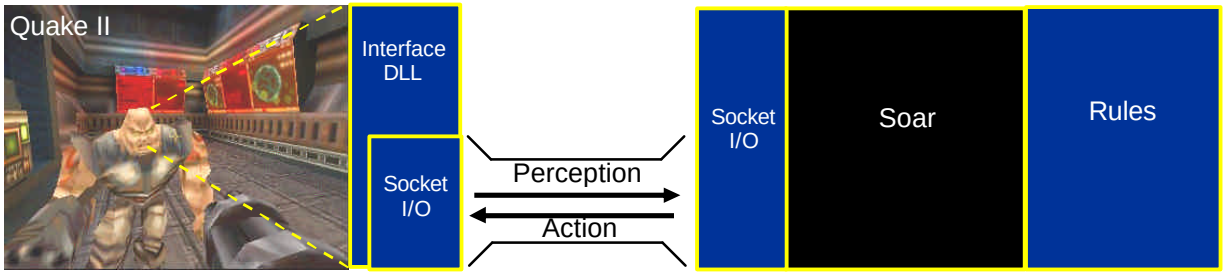
Part VI: Building Large Soar Programs: Soar Quakebot

This part of the tutorial describes the simple version of the Soar Quakebot, a Soar program that plays Quake II against human players in deathmatches. It provides an example of a moderately large Soar system. The version covered in this tutorial is simpler than the full Quakebot, which also includes mapping and anticipation capabilities. That version is available, but the shear size of it (over 800 rules) makes it difficult to understand. This tutorial also provides an introduction to a Quakebot that others can use to develop their own Soar Quakebot. More generally, it provides a detailed exemplar of how Soar can be used to develop AI entities for real computer games. The goal of our Soar/Games project (ai.eecs.umich.edu/~soarbot, ai.eecs.umich.edu/people/laird/gamesresearch.html) is to use Soar as an engine for many different genres of computer games.

This part is less of a tutorial and more overview documentation of the Soar Quakebot. It does not cover in detail the all of the rules used in the system. The Soar Quakebot has over 300 rules, so that the purpose of this tutorial it to go over the structure of the Quakebot's operators and the underlying data structures. I highly recommend that you use Visual-Soar to go through the operators, rules, and state structures while reading the tutorial. For a technical examination of the Soar Quakebot with emphasis on its ability to use anticipation (not covered in this tutorial) see my paper from the AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment: "It Knows What You're Going to Do: Adding Anticipation to a Quakebot" (ai.eecs.umich.edu/people/laird/papers/anticipation-print.pdf).

1. The Quake II Environment

The current version of the Soar Quakebot runs on a separate computer from Quake II and you must buy your own copy of Quake II. Soar interfaces to Quake via Socket I/O, which resides on both computers. The two parts of Socket I/O communicate using working memory element-like structures. The Interface DLL loads into Quake II and contains all of the code that accesses the Quake II functions and data structures for simulating a Quakebot's perception and performing its actions. The DLL uses Socket I/O to send input-link data structures to and receive the output-link action commands from Soar. The advantage of Socket I/O is that it provides a task-independent interface for connecting Soar to other applications. Only the code in the DLL must be changed (and the rules) when using Soar in another applications. The details of the interface are completely hidden from the Soar side, with the Soar program getting input via the input-link and sending commands to control the Quakebot via the output-link. Unfortunately the current implementation cannot run both Soar and Quake II on a single computer. We are developing a new vision that will allow multiple Soar entities to run on the same computer as Quake II.



1.1 Installing Quake II, and the Soar Quakebot

Not yet available
The installation instructions for the Soar Quakebot can be downloaded from http://ai.eecs.umich.edu/~soarbot.

1.2 Playing Quake II

Our Quakebot plays the death match version of Quake II. In a death match, players exist in a "level", which contains hallways and rooms. The players can move through the level, picking up objects and firing weapons. The object of the game is to kill the other players as many times as possible in a given amount of time. Each time a player is shot or is near an explosion, its health decreases, and when the player's health hits zero the player dies. A dead player is "respawned" at one of the specific spawning sites within the level. Throughout the level are "powerups": weapons, health, armor, and ammo. Success in the game depends on collecting these during a fight. When one of these objects is picked up (by moving to its position) a new one will automatically appear in the same position in 30 seconds. When a player dies, the currently selected weapon (except for the blaster), will appear near the body, but will disappear within 30 seconds if it is not picked up.

Weapons vary according to their range, accuracy, spread of damage, time to reload, type of ammo used, and amount of damage they do. For example, the shotgun does a lot of damage in a broad area if used close to an enemy, but does little or no damage if used from a distance. In contrast, the railgun can kill in a single shot at any distance, but requires very precise aim because it has no spread. For more details of Quake II, see sections 1.4-1.6 below and the Quake II manual.

1.3 Soar Quakebot Limitations

There are a number of limitations in the current Quakebot, some from the overall structure or our system and some from limitations on the reasoning of the Quakebot. These are listed below:

1. Only one Quakebot fights one human at a time. This is a limitation of our overall software structure. We are working on changing this so any number of Quakebots can fight any number of humans.
2. The more advanced Quakebot can map and navigate only two-dimensional levels that are made up of rectangular rooms connected by hallways. This restriction comes from the method it uses to build an internal map the level. The fighting, wandering and other behaviors use the internal map but could be extended to work in more complex levels if the bot had a way to map it. The simple Quakebot does not use a map, so it is more flexible, but less intelligent.
3. The Quakebot cannot negotiate lava or water.
4. The Quakebot does not crouch, and it aims only in two dimensions.
5. The Quakebot does not know how to use grenades.
6. The Quakebot does not know about buttons or switches.

1.4 Soar Quakebot Sensors

Each Quakebot continually sees and hears things in its environment. It has some additional sensing abilities relating to distances from walls. It also senses aspects of its own state, aspects of the game, and feedback from its actions. These structures are described in http://ai.eecs.umich.edu/~soarbot/ilink.html. Below is a discussion of the most important aspects of the sensors. Each of these is an augmentation of the input-link: (`state <s> ^io.input-link <il>`).

1.4.1 Game Data

The only game data currently available is the name of the current level, which is a string called the mapname: (`state <s> ^io.input-link.game.mapname <string>`). The name is defined when the level is created using the level-editing tool and it is used by the Quakebot to retrieve the correct map for the current level (assuming it has a map built for this level).

1.4.2 Agent Data

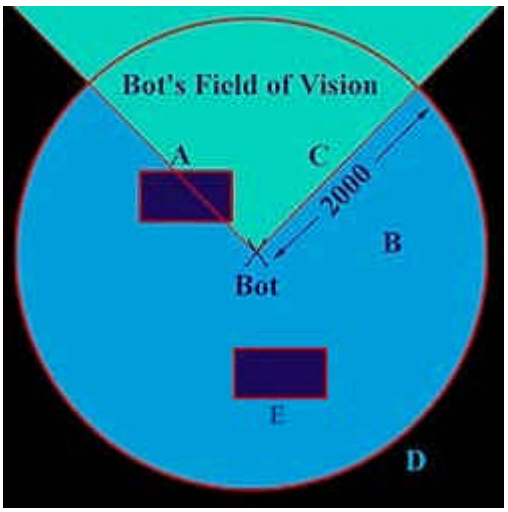
Agent data describes all of the data about the bot: (`state <s> ^io.input-link.agent <agent>`). It includes information on the bot's name, as well as its current position, orientation, velocity. It also includes information on its health, armor level, whether it is dead or alive, which items it has, which item is selected, which weapons it has, and which weapon is selected. The agent data also has information on the current decision cycle and a random number that is updated every cycle and can be used by rules to make random decisions (such as how long to camp out waiting for the enemy).

1.4.3 Entity Data

Sensory data for each visible entity in the game is created on: `(state <s> ^io.input-link.entity <entity>)`. A Quakebot can sense all unobstructed powerups that are in its field of view. The same goes for enemies and projectiles in flight. The diagram below shows which object a bot can see and its field of view. In the figure there are two obstacles, one to the front-left, and one behind, and five objects (A-E). The bot can see only object C. All of the others are either out of range (D), not in its view cone (B & E), or occluded by an obstacle (A).

Status of the targets:

target	inrange	visible	infront
A	Yes	No	Yes
B	Yes	Yes	No
C	Yes	Yes	Yes
D	No	N/A	N/A
E	Yes	No	No



For each powerup that the bot sees, it senses the object's type, exact location, and its position relative to the bot. For each enemy, it also senses its orientation, its health and death status, its name, velocity, and currently selected weapon. For projectiles in flight, the Quakebot senses its type, its position relative to the bot, and whether it is headed toward the bot and likely to hit. The sensing of projectiles is not yet used by the Soar Quakebot (it seems to still have a few bugs in it).

1.4.4 Sound Data

A Quakebot can hear sounds, such as the human running nearby and explosions even through walls. For each "sound", the bot gets the relative position of the sound and the type. The running of other players is masked if the bot is running, but not if the bot is stopped or walking.

1.4.5 Map Data

The most difficult sensing for a bot is to sense the walls that make up the level. Within Quake II (and similar games), there are no rooms with 3" thick walls. Instead there are thousands to millions of infinitely thin polygons that when displayed give the illusions of walls. The information available to the bot is the range from the bot to the nearest walls in front, behind, up, down, to the left, and to the right of the bot. The bot also gets range information a few degrees to the left and right of its front. This information is used to detect when it has caught its shoulder on a doorway, where the front sensor will show that it is clear, but the bot cannot move forward.

1.4.6 Feedback Data

In contrast to Eaters and TankSoar, the Quakebot does not receive feedback on its actions through augmentations on the output-link associated with its actions. The Quakebot receives all feedback on input-link. This should be changed at some point.

1.5 Soar Quakebot Actions

The Quakebot has a variety of actions including moving through the world (thrust forward/backward/stop, run/walk, turn left/right, sidestep left/right), attacking other players (attack, face-enemy/lead-enemy), and selecting weapons.

1.6 Summary of Input-link and Output-link

The details of these structures are in http://ai.eecs.umich.edu/~soarbot/ilink.html.

2. Soar Quakebot Overview

The Soar Quakebot's knowledge and goals are organized hierarchically; similar to the way they were organized in TankSoar. At the top-level state there are a combination of goals (ambush, attack, chase, collect-powerups, explore, hunt, predict-enemy, retreat, face-attack, and wander), and primitive operators (die, spawn, record-enemy, select-enemy, remove-enemy, record-sound, remove-sound). Shared implementations of operators that occur in multiple goals are in "common." Finally, rules used to elaborate the current situation at the top level are found under "elaborations." The following description of the Quakebot knowledge and goals is organized by activity, such as wandering, living and dying, building and maintaining the map, collecting powerups, managing information about the enemy, pursuing the enemy, attacking the enemy, predicting the enemy's behavior. The most complex parts of the Quakebot involve building up the map; however, the map structure is by itself straightforward and is used by almost all other activities. Therefore, although building the map is best put off to the end (and so far has been put off completely), an overview of the data structure is best done before describing specific activities. The activities themselves are presented in order of complexity, from simplest to most complex.

Although most of the operators in the Quakebot are organized hierarchically in terms of goals and subgoals (or tasks and subtasks), there are some operators that can be selected at any level of the operator hierarchy. These are often called "floating" operators and are kept in the all/ directory. These operators perform tasks that need to be done no matter what top-level operator is being attempted, such as selecting the best weapon, noticing that the bot just moved into a room, noticing the location of powerups in the room, getting the bot unstuck, and noticing when the bot's health or armor decreases.

Below is a list of the main tactics the simple Quakebot uses. These are implemented across the top-level operators.

- Collect-Powerups
  - Pick up best weapons from spawn locations
  - Remember when missing items will respawn
  - Use shortest paths to get objects
  - Get health/armor if low on health/armor
  - Pickup up other good weapons/ammo if close by to deny enemy
- Attack
  - Use circle-strafe
  - Move to best distance for current weapon
- Chase
  - Go after enemy based on sound of running
  - Go after where enemy was last seen
- Ambush
  - Wait in a corner that can’t be seen by enemy
  - Wait by a door where an enemy is expected to come out
- Hunt
  - At nearest spawn room after killing enemy
  - At rooms enemy is often seen in

3. Map Data Structure

Much of the behavior of the Quakebot is driven by its knowledge of the map, which is built up using the explore operator. The map is an augmentation of the top state. Some of the augmentations that are part of the map data structure are used only during the creation of the map. Once exploration is complete, only a few aspects of the map data structure are modified (such as the pointer to the current room). In this section, only those parts of the map that are used after the map is created will be described. The explore operator is not described in this tutorial.

3.1 Map Overview

The map consists of a set of room objects, with each room having wall and door objects. The map is an augmentation of the top-state (<s> ^map <m>). The map has the following augmentations (there are many more map augmentations that are used during exploration to build the map, but these are the most important ones).

```
room              A room in the level (there is a separate augmentation for
                  each room in the level).
current-room      The room that the bot is currently in.
last-room         The last room that the bot was in.
enemy-room        The room that a sensed enemy is in (if not the current
                  room).
explored [true]   Signifies that the map has been completely explored.
item              A powerup that is noticed in the level during exploration.
```

The initialize-bot operator creates the map augmentation on the top state and other map related augmentations.

The current-room is computed by an elaboration in elaborations/map.soar. It continually monitors the Quakebot's current x, y location and finds the room with walls that enclose the Quakebot. This calculation only works once the map of a room is created, so during exploration the Quakebot relies on a more deliberate calculation of its current-room.

3.2 Rooms & Doors and Walls

3.2.1 Rooms

Each room has many augmentations. Below are the most important ones:

```
type [hall/room]  The type of room. A hall connects rooms together.
room-number       An integer that uniquely identifies this room.
door              One for each door in the room.
at-door           The door that the bot is at (if it is at a door).
wall              One for each of the walls that make up the room.
resurrect         Contains the x, y location of a spawn location in the room.
hiding-place      Contains the x, y location of a hiding place in the room.
                  Calculated by rules in elaborations/hiding-place.soar
search            Temporary information that is maintained about the current
                  visit to a room - more details given below.
item              A powerup that was found in the room during exploration.
enemy-visit-count The number of times an enemy has been seen in this room.
path              Path information for this room to every other room. This
                  structure contains the distance to the other room and the
                  doorway to go through that leads to that room.
```

3.2.2 Doors

Below are the most important augmentations of each door object.

```
connecting-door   The door object in the other room that connects to this
                  door. A door is specific to a room.
direction         The cardinal side of the room that the door is on:
                  north, south, east or west.
number            The number of the door - each door has a unique number.
room              The room that this door is in.
wall              The wall that this door is part of.
x                 The x location of the middle of the door.
y                 The y location of the middle of the door.
```

3.2.3 Walls

Below are the most important augmentations of each wall object.

```
door              One for each door in the wall.
side              The cardinal side of the room that the wall is on:
                  north, south, …
x                 If the wall is east or west, the x location of the wall.
y                 If the wall is north or south, the y location of the wall.
```

3.3 Search

A search object is created each time a room is entered and is deleted when the room is exited. It contains temporary information that is useful during the visit to that room. The all/enter-room.soar operator creates the search data structure whenever a room is entered, deletes the search data structure from the previous room, and updates the last-room augmentation of the map (so the Quakebot can avoid going back to the previous room when it is wandering).

```
hide-time         The length of time to hide in a room.
recorded          Information on an item that was noticed in the room during
                  this visit to the room.
```

3.4 Items

All the powerups in the game, such as health, armor, weapons, and ammo, are called items. General information on each type of item (ammo, armor, health, and weapons) is added to the parameters augmentation of the top-level state by rules in elaborations/ammo, /armor, /health, and /weapons. An item that is seen by the bot will be recorded as an ^item augmentation of a room. Each item will have the following augmentations:

```
available [true]  This is created if the item is expected to exist - it has
                  not been detected to be missing.
classname         The classname of the item.
regeneration-time The time when the item is expected to be regenerated if
                  not currently available.
room              The room the item is in.
type              The type of the item: health, armor, weapons, or ammo.
x                 The x location of the object.
y                 The y location of the object.
```

Operators create the item data structures. During exploration, all/notice-item and all/record-explore-item creates item augmentations that the Quakebot notices in rooms. Notice-item creates an initial item augmentation of the map, and record-explore-item records it in the room once the room's walls are known. These items are powerups that are part of the map and the bot remembers their locations. During other top-level operators, such as wander, chase, collect-powerups and others, the Quakebot notices

items in rooms it visits and records them in the room search data structure by all/record-temp-item. It also notices and records weapons that have been dropped by players when they are killed.

There are two additional operators that maintain the available and regeneration-time augmentations. The all/notice-item-missing operator is proposed when the Quakebot is attempting to get an item, is in the room where the item is supposed to be, is facing where the item should be, but does not see the object. The Quakebot sets the regeneration-time, which is when it will consider that the object is available. However, the regeneration-time that the Quakebot creates is only an estimate (because the Quakebot does not know when the item was picked up) and so the all/notice-item-present operator is selected when an item was thought to be missing, but is seen to exist. This operator resets the regeneration time, which in turn makes the item available. The calculations for whether an item is available are done by rules in elaborations/items.soar.

4. Quakebot Self

The Quakebot maintains a lot of information about itself as it plays the game. Many of these are on the self augmentation of the top-state, which is created in elaborations/self.soar.

4.1 Movement

One of the nasty problems in controlling the Quakebot is that it is hard to tell when it gets stuck, such as catching its shoulder on the edge of a door, which prevents it from moving. The only indication that the Quakebot is stuck is that it is trying to move and it is not moving. Unfortunately, the calculation of being stuck cannot be done immediately once the Quakebot starts to move because it takes time for the Quakebot to start to move. The Quakebot uses the following augmentations and operators to detect if it is stuck and attempt to become unstuck.

self.stopped [true]	This augmentation is created by a rule in elaborations/self.soar when there is no velocity in any direction.
self.stuck [left right]	This augmentation is created by a rule in elaborations/self.soar when the Quakebot is trying to thrust forward or backward, but the Quakebot is stopped and a sensor (leftlos/rightlos) indicates that there is an obstruction to one side, but none in front.
self.stuck-time	This augmentation is created by the all/record-stuck operator. This operator is selected whenever the bot is stopped. It records a time in the future when the Quakebot should have had enough time to get moving. This augmentation is removed by the all/remove-stuck operator if the bot is no longer stopped.

If the bot is stuck left or right and the current time is greater than stuck-time, then the all/sidestep operator is proposed. This operator causes the bot to thrust backward and away from the side it is stuck. Once the bot is no longer stuck, other operators should take over to make it go forward along its original path.

If the bot is stopped but is not stuck and is not waiting for an ambush, the operator all/start-moving will be proposed, which will move the bot forward.

4.2 Wants and Needs

The Quakebot continually maintains a list of the powerups it thinks it wants (these are powerups that would be good to have but it won't actively pursue them) and needs (it will actively pursue these). These augmentations are created on the ^self augmentation of the top-state in elaborations/self.soar.

For health, the bot has predefined parameters (set as augmentations of the top-state.parameters) for the level of health it has to have before it wants or needs more health. Similarly, it wants armor if its level of armor is below some level. It also wants and needs ammo for its current gun. Finally, if there are better weapons than the one it has, it will need those weapons.

The needs trigger the creation of a get-object augmentation of the self object if the bot thinks that there is an item of the type needed available. The get-object differs from a need because it is instantiated with a specific item that the bot knows about. The get-object then triggers the proposal of the collect-powerups operator.

4.3 Retreat Information

Elaborations in elaborations/self.soar also recognize when the bot should attempt to retreat, either because it has low health or is outmatched in terms of weapons. Even when the bot wants to retreat, the retreat operator will be proposed only if the bot is near and exit and not near the enemy (otherwise it is too easy to get shot).

4.4 Health & Armor

One shortcoming in the sensors for the bot is that it does not get any direct indication when it is hit by an enemy's weapon. This is only important when the bot does not sense the enemy directly. Thus, the bot must notice when its health or armor suddenly decreases from a shot in the back. Then it can turn and face the enemy (and the normal attack operators will take over).

In order to notice that the health or armor has decreased, the bot records its initial health and armor on the self augmentation as self.saved-health and self.saved-armor using a rule in elaborations/self.soar. Whenever the bot's health or armor changes, the operators all/update-health and all/update-armor update the saved values. However, if the current value is less than the stored value and there is no enemy detected, then the face-attack operator is proposed, which causes the bot to turn 180 degrees (and should cause it to see its enemy).

4.5 Current Weapon

Quake automatically switches to what it thinks is the best available weapon when a new weapon is picked up. However, in some cases, the bot might have different priorities and all/select-weapon ensures that the selected weapon is the one the bot thinks is best.

4. Die and Spawn

Sometimes, probably through a lucky shot by the human, the bot will be killed. In Quake, after a player is killed, it is "spawned" at one of a few specific spawn locations on the current level. The bot gets information over the input-link when it dies (io.input-link.agent.deadflag dead). When it dies, the top-level die operator is selected. Once die is selected, the bot explicitly stops all actions (turns off attack, sidestep, …), and marks that it is dead (<s> ^died true). It also removes the last-room structure, any retreat structures, and any enemy structures.

When the bot is spawned, which it detects because it thinks it is dead (<s> ^died true), but the input-link indicates it is alive (io.input-link.agent.deadflag dead), the spawn operator is selected. The spawn operator resets the last-room to none, removes the (<s> ^died true) and records the spawn location if it is not already recorded. The bot uses the spawn location in one of its tactics.

5. Enemies

A critical part of winning in Quake II is keeping track of enemies. Enemies can be seen visually, but when they go out of view, they no longer come in via the input-link. The interface does preserve the "image" of an enemy for 2 seconds after they disappear. That is enough time to select an operator to explicitly remember the last position of the enemy. Other important aspects of enemies is that they can be heard if the bot is not running. The sound sensor gives the bot a direction and range to the sound (as if the wall were made of paper). The maximum range of the sound sensor is 1000 quake units.

In the remainder of this section, the details of the operators for selecting, recording, and forgetting enemies. These descriptions also cover the enemy data structures.

5.1 Select-enemy

The select-enemy operator is proposed when an enemy is detected and there is not already an enemy selected. A rule in elaborations/enemy.soar recognizes when there is a live enemy that is visible and in front of the bot and augments the top-state with (<s> ^enemy-detected <e>), where <e> is the enemy object on the input-link. If the enemy is in front of the bot, select-enemy will immediately start shooting, otherwise, attacking the enemy is left up to the attack operator. The select-enemy operator is preferred to most other top-state operators so that the bot can respond to the enemy ASAP. One exception is if the enemy is not facing the bot and the bot has only the blaster and the enemy has a better weapon. In that case, the bot will attempt to avoid the enemy. If there is more than one enemy, the bot will select the closest enemy.

Once the select-enemy operator is selected, the bot creates an enemy object on the top-state, which in turn has an augmentation (sensed-enemy) that points to the enemy data structure on the input-link. The select-enemy operator also increments a counter for the number of times it has seen the enemy in the current room. This count is used to predict where the enemy likes to hang out. After select-enemy is applied, the attack operator will normally be selected.

5.2 Record-enemy

If the enemy disappears (but does not die), then the record-enemy operator is proposed. It is preferred to most other top-state operators. When it is selected, it turns off shooting and augments the enemy data structure with all of the information it last perceived concerning the enemy. In addition, a future time is recorded for when this record is obsolete.

The attack/follow-enemy operator will then be selected to move to where the enemy was last seen.

5.3 Forget-enemy

If the record of the enemy becomes obsolete, or an enemy dies, then the forget-enemy operator is proposed. It is preferred to most top-state operators. This operator removes the enemy augmentation from the top state and turns off shooting. It also removes any retreat augmentations and creates an augmentation on the top-state that the enemy-just-died. This augmentation cues the proposal of the hunt operator.

5.4 Record-sound

If an enemy is not detected and a sound is heard, the record-sound operator is proposed. It is preferred to most top-state operators. The action of this operator is to create a sound object on the top-state that includes the type, range, and direction of the sound. A future time is also recorded for when the sound will be obsolete. The sound object cues the proposal of the chase operator.

5.5 Forget-sound

This operator is selected when a sound becomes obsolete. It removes the sound object.

6. Wander

The wander operator is the default operator that is selected when the Quakebot has nothing else to do (explore, attack, chase, ambush, …). It makes the bot wander from room to room by going through a door in the current room using go-through-door as its only suboperator. If the bot sees a powerup or the enemy, another top-level operator will be selected (such as select-enemy or collect-powerups). While wandering, the bot walks so it can hear the enemy running nearby.

In proposing go-through-door, if there is only one door in the room, the bot will go through that door. If there is more than one door, the bot will not go back to the room it just came from (saved in last-room). If there are multiple doors to go through, the bot will prefer the door that leads to the room where it has seen the enemy the most (assuming it has seen the enemy there at least enemy-visit-cutoff number of times).

Although go-through-door is the only explicit suboperator of wander, other operators from the all/ folder may be selected during wander. Specifically, enter-room will be preferred each time a new room is entered. This operator updates last-room and initializes the search object for the current room (and removes the search object for the previous room).

7. Collect-powerups

To win in Quake II, the bot should try to stay healthy and have the best weapon. The collect-powerup operator takes the bot to the powerups that the bot has seen during exploration and picks them up. It is proposed when there is some item that the bot needs (see Section 4.2). In general, all other top-state operators are preferred to collect-powerups except for wander.

There is only one suboperator: get-item. Get-item is proposed for every item that the bot needs and any item that the bot wants that it sees in the current room. Get-item is also proposed for ammo or good weapons if they are close (parameter.close-ammo-range, parameter.close-weapon-range) even if they are not needed or wanted. The idea is to deny them to the enemy and to have backup weapons and ammo if they are needed in the future.

There are many selection rules (in common/get-item.soar) for deciding which item to pursue.
1. Prefer items that are close within the current room to items not in the current room.
2. Prefer items with higher priority (overridden by 1 above)
    - Weapons = priority 7
    - Health & Armor = priority 6
    - Close Ammo = priority 4
    - Far Ammo = priority 3
3. For non-weapons, prefer items that are closer if they are of equal priority.
4. For weapons, prefer the better weapon.
5. For equal weapons, prefer the weapon that is closest.

The distance calculations are based on actual distance if the items can be seen, or number of rooms to go through for items in different rooms.

8. Retreat

The retreat operator is only selected if there is a retreat structure on the self object (built by rules in elaborations/self.soar) and the enemy is either far away (> parameter.retreat-range) or not visible. The retreat operator attempts to run away from the enemy by going from room to room, avoiding returning to a room it has been to during the retreat. The record-visited-room operator adds each room that is visited to the retreat structure and then the go-through-door operator takes the bot through a door. The closest door to the bot is preferred, except if the door takes the bot to a room it has been to during the retreat.

9. Chase

The chase operator is proposed if the Quakebot has recorded a sound, does not see an enemy, does not need health, and has a weapon that is better than the blaster. This operator is better than wander, explore, and collect-powerups. This operator will only be selected after record-sound and will be displaced when the sound becomes obsolete by the selection of the remove-sound operator. Once the operator is selected, there is an application rule that will update the sound objects whenever there are any newly sensed sounds.

There are two suboperators for chase: turn and go-through-door. Turn is selected if the bot is not facing the sound. It will turn the bot toward the sound. If this leads the bot to see the enemy, select-enemy will displace chase, followed by attack. If no enemy is seen, then the bot will attempt to go to a room that is in the direction of the sound using the go-through-door operator. The bot calculates the wall that the sound is coming through and if there is a door on that wall, will go through it. If there are multiple doors, the bot randomly selects one. If there is no door on that wall, the bot randomly selects a door from among those that are on walls next to the wall with the sound, otherwise the bot exits through a door opposite the sound.

10. Attack

The attack operator is proposed when an enemy has been selected. Once the attack operator is selected, elaborations rules dynamically decide which type of attack to use. The Quakebot will use a direct attack if the enemy is not facing the bot, otherwise it will use circle strafe. The following operators are used to implement the attacks.

9.1 Select-target

Select-target sends a command to the Quakebot control system that the current target is the selected enemy. This is used in medium and good aiming skill.

9.2 Face-enemy

Face-enemy uses three different techniques to aim at the enemy. None of these handles the third dimension (I just haven't had time). The one used is determined by the aiming-skill parameter.

- Bad: The bot explicitly turns to face the enemy. There are no calculations to lead the enemy nor taken into account the bot's movement.
- Medium: The bot uses a C-coded routine to continually face the enemy. This is faster than having the bot do it, but it still does not lead the enemy.
- Good: The bot uses a C-coded routine to lead the enemy. This should be really tough but still misses when the enemy or the bot moves.

9.3 Approach-enemy

Each weapon has a desired range of distance for use. Approach-enemy uses information from the currently selected weapon to either approach or move away from the enemy. This could be refined by looking at the enemy's weapon and picking a range that is best for the bot's weapon and worst for the enemy's.

9.4 Charge-enemy

The charge-enemy operator is selected if the enemy is in a different room and there is no reason to retreat. The bot will charge forward toward the enemy. Not clear that this is a good tactic, but it keeps the bot engaged in the fighting and helps avoid losing track of the enemy.

9.5 Direct-Attack

If the enemy is not facing the bot, then the bot will stop sidestepping. This operator will move the bot forward to get it to the mid-point of its optimal range. Approach enemy just tries to get it within the maximum and minimum of the optimal range.

9.6 Circle-strafe

If the enemy is facing the bot, then the bot will circle strafe by continually sidestepping to the left or right will aiming at the enemy. This makes it harder to hit the bot. Circle strafing will randomly switch directions based on a timer and the bot also switches directions when it gets near a wall.

9.7 Get-item

Get-item allows the bot to pick up items that it needs while it is fighting. Get-item is selected only if the object is close (parameters.attack-get-item-range) to the item.

9.8 Shoot

When the bot is either tracking the enemy, or is facing the enemy, it will start shooting.

9.9 Jump

If the enemy has the rocket launcher or the railgun, the bot will randomly jump.

9.10 Clean-up-enemy

If there is a recorded enemy structure that is obsolete, then delete it. This is important so that the bot doesn't go after an old position of the enemy when it should be engaging an active enemy.

11. Predict-enemy

The predict-enemy operator attempts to predict what the enemy will do next. The result of this operator (a prediction structure added to the enemy object) is then used by ambush, hunt, and deny-powerups to defeat the enemy. The predict-enemy operator is proposed when the enemy is no longer visible, or is facing away from the bot and is not too close. The idea being that in cases where the enemy is running away it might be better to ambush the enemy than to try to run him down from behind. This will also work on levels where the bot can see the enemy far away, but cannot immediately attack the enemy (we will have to build some levels like that!).

Predict-enemy works by creating an internal state that appears to be just like the top-state but has the information in it from the enemy's perspective (create-enemy-state). This information will be incomplete, but sufficient to do some prediction. The bot then uses its own knowledge of tactics to predict what the enemy will do next by just letting its operators get selected. Some additional knowledge must be added so that the prediction moves forward through operator applications. This is done by rules that short-circuit operators, such as go-through-door, by mentally tele-porting the bot through the door by changing internal objects (predict-enemy/nternal-simulation.soar). During the simulation, the bot also keeps track of how far the enemy goes in terms of number of rooms (also predict-enemy/internal-simulation.soar). When the simulation gets to the point where either the distance traveled by the enemy will be more than the distance that the bot needs to travel to get to get to the same location, or where there is indecision for the enemy, then the simulation terminates, with the result being the last room the enemy will be in and the door it went through to get to that room.

12. Hunt

The hunt operator is selected when the bot has a prediction as to which room the enemy will be in. One prediction comes from predict-enemy and the other when it has just killed the enemy - it predicts the enemy will be in a spawn room. Overall, hunt heads toward the room that it predicts the enemy will be in. If it is already in a spawn room that it predicts the enemy will be in, it turns to the spawn point. Once at the spawn point, the bot will forget about the enemy if the enemy is not there.

13. Ambush

The ambush operator is selected under two distinct circumstances.

If the bot is in a room with a hiding place, it will propose ambush. The ambush operator will be selected only if there are no other more important operators to be selected, such as attack, collect-powerups, … For this type of ambush, the bot goes to the hiding place and then turns and faces out into the room (ambush/hide-in-corner)

If the bot has a prediction as to which door an enemy will come through for this room, the bot will hide off to the side of the door (hide-by-door). The bot will select a hiding place (pick-hiding-place), move to that hiding place (move-to-hiding-place), and then face the door where it expects the enemy to come through (move-to-hiding-place). I should add that if there is sound and the bot has the rocket launcher or a grenade, it should shoot into the hall before it sees the enemy.

For all of the ambushes, the bot determines a time to wait and will give up the ambush after that amount of time.

14. Deny-powerups

If the enemy is predicted to go after a specific item, the bot will attempt to go to the room where the item is and get it before the enemy does.