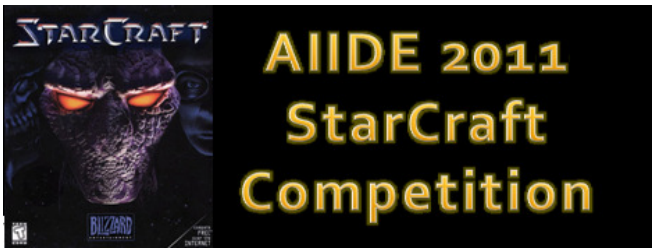
1 Glossary

1.1 AIIDE 2011 StarCraft AI Competition



The competition (http://eis-blog.ucsc.edu/2011/02/aiide-2011-starcraft-ai-competition/) is organized in parallel with the AIIDE 2011 conference. About thirty teams are registered. The goal is to build an agent AI that can win against the other competitors. No cheats are allowed, fog of war is on, and AI agents are disqualified if they crash the game (hehe). Source code must be submitted along with the agents at some point in August.

1.2 BWAPI

BWAPI (http://code.google.com/p/bwapi/) is an API allowing one to retrieve info and send commands to the StarCraft executable. The AIIDE competition will use the latest BWAPI version.

1.3 BWTA

BWTA (http://code.google.com/p/bwta/) is an add-on library to BWAPI taking care of terrain analysis. It identifies areas, chokepoints, possible starting locations, etc. It is used by BWSAL (and also by our SPAR agent).

1.4 BWSAL

BWSAL (http://code.google.com/p/bwsal/) is another add-on library to BWAPI (but dependent on BWTA) implementing a vast array of managers taking care of various tasks, from resources allocations to unit productions to buildings placement, but not combat management. BWSAL is meant to facilitate the implementation of AI agents. Although AIIDE competitors are not required to use BWSAL, our SPAR agent is inspired in part by some of these managers, particularly in the "Actions implementation" (layer 2) of the decision making processes.

1.5 SPAR

SPAR (System for Planning and Adversarial Recognition - http://planiart.usherbrooke.ca/projects/spar/) is the (cool) name of an architecture developed in the Planiart lab. By extension, it is also the name of the StarCraft agent we are currently building using this architecture.

2 Solution

The AIModule solution includes more or less all development code related to StarCraft.

2.1 BWAPI

Header files for BWAPI library. The BWAPI debug and release libraries can be found in the /lib folder. The entire code (including source files) is stored in the following SVN repository: http://bwapi.googlecode.com/svn.

2.2 BWTA

Header and source files of the BWTA library modified for the SPAR agent. The original BWTA code can be found in the following SVN repository: http://bwta.googlecode.com/svn.

2.3 BWSAL

Header and source files for the BWSAL add-on library, modified for the SPAR agent. The original BWSAL code can be found in the following SVN repository: http://bwsal.googlecode.com/svn.

2.4 MacroAIModule

Example of an incomplete AI agent shipped with the BWSAL library. Not necessary for compiling the SPAR agent.

2.5 PlanRecognitionAIModule

Incomplete AI agent sending observations (through sockets) to a Java module performing plan recognition. Not necessary for compiling the SPAR agent.

2.6 TeamStarcraftAIModule

An AI module developed by Steve Tousignant, Anthony Jo Quinto et Frédéric St-Onge during the Winter 2010 term for the IFT702 and IFT592 courses. The AI took second

place (out of seven competitors) in the second challenge involving micro-management of units. Documentation is included. We would like to design behaviours (layer 1 of decision making) inspired by their work. Not necessary for compiling the SPAR agent.

2.7 SparAIModule

Static library for the SPAR AI agent which includes all code related to both the architecture and the AI agent itself. The SPAR AI plays Protoss, we assume the game is 1vs1 with one Nexus and four Probes as starting units.

Sub-folders are as follows:

2.7.1 DecisionMaking

Includes the 4 layers of decision making. All code related to that should go in the appropriate sub-folder.

2.7.2 PerceptualState

Includes structures used to reason about the terrain, the agent units and the enemy units.

2.7.3 Scenarios

Includes code related to scenarios. Scenarios are usually meant to test a particular component of the AI, without interference from the other components. All scenario-dependent code should go there.

2.7.4 Scheduler

Schedules the various tasks to different frames so that the agent does not slow down the game while playing.

2.7.5 SituationAnalysis

“Situation analysis” should be understood as having the same meaning as “sensing”. So this folder includes the 4 layers of sensing. All code related to that should go in the appropriate sub-folder.

2.7.6 Utils

Common code shared by all components of the SPAR agent.

2.8 SparQtGUI

Static library for a graphical user interface prototype for the SPAR agent, made in Qt.

2.9 SparAIModuleDLL

This project creates the DLL library that instantiates the SPAR agent and the GUI if needed.

2.10 Utils

Common code shared by several projects.

2.11 Start_ChaosLauncher

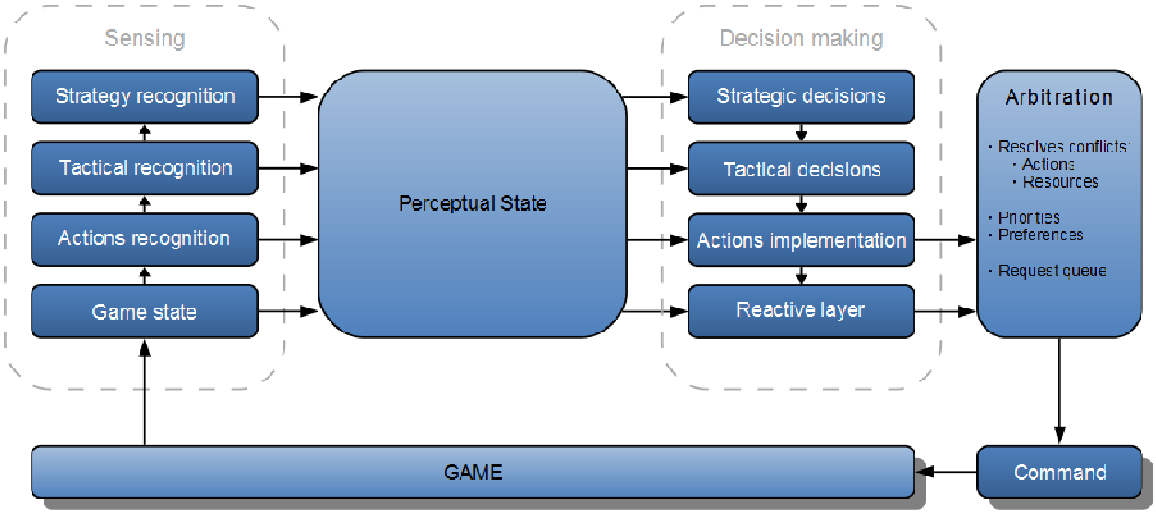
Automatically launches ChaosLauncher (which, in turn, should launch StarCraft) and then attaches the debugger to the StarCraft executable.

2.12 Documentation

Generates documentation (web pages) from the SPAR agent source code annotations using Doxygen.

3 Architecture (SparAIModule)

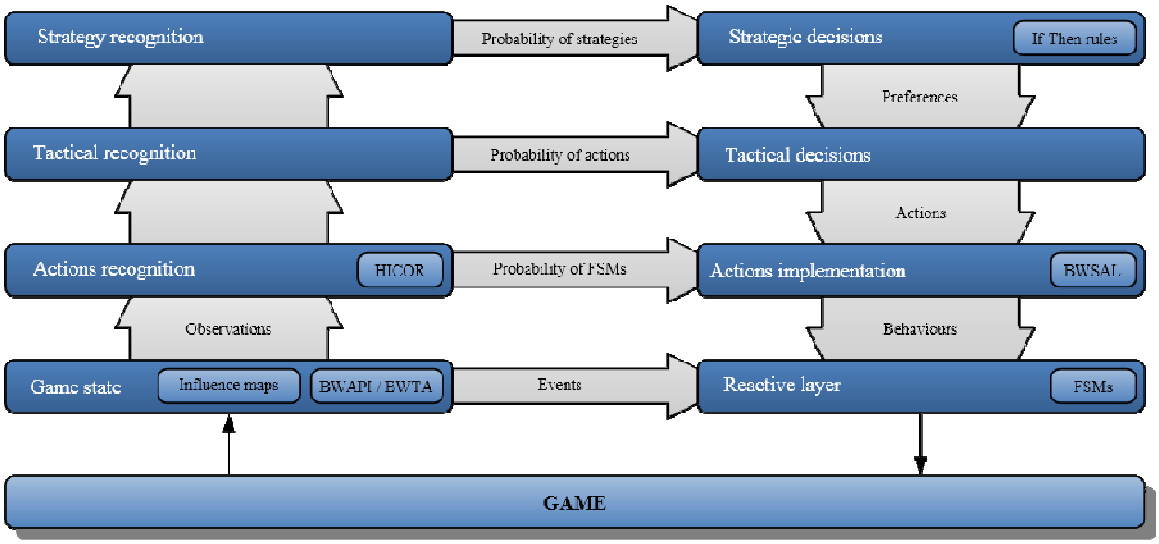
Here's a high-level view of the architecture:



An idealist (non-practical) view of the information flow is as follows:

1) Information is retrieved from the game state;
2) The sensing modules refine the information and output the refined stuff in the perceptual state;
3) The decision making modules retrieve the info from the perceptual state and take decisions;
4) These decisions modify the game state.
5) Loop back to 1).

The following image details the (again idealist) view of the plan recognition processes:



3.1 Sensing

Not much work has been done on sensing to date, so the descriptions in this section are to be considered as objectives, not actual functionality.

3.1.1 Game state

Gives the low-level information about the terrain, the units (e.g., position) and so on.

3.1.2 Actions recognition

Recognizes the individual actions currently hypothesized to be pursued by the opponent, e.g. a group of units attacking one of our bases.

3.1.3 Tactical recognition

Recognizes the short-term tactical plans hypothesized to be executed by the opponent, e.g. a group of units attacking one of our bases to draw defenders away from another position.

3.1.4 Strategy recognition

Recognizes the mid- and long-term objectives hypothesized to be pursued by the opponent, e.g. saving up resources and larvae until the Spire (aerial creatures facility) is completed in order to hatch Mutalisks (aerial creatures) immediately after.

3.2 Perceptual State

The perceptual state stores (more or less refined) data supplied by the sensing modules. In particular, it includes various types of influence maps quantifying the level of threat, importance, or some other metrics per area. It also manages a representation of the current map as a graph with vertices being locations and edges being chokepoints (see BWTA).

3.3 Decision making

3.3.1 Strategic decisions

Figures out mid- and long-term objectives acting as search-control, for example, favour getting anti-air defence if an enemy Spire has been spotted.

3.3.2 Tactical decisions

Computes short-term abstract actions to execute.

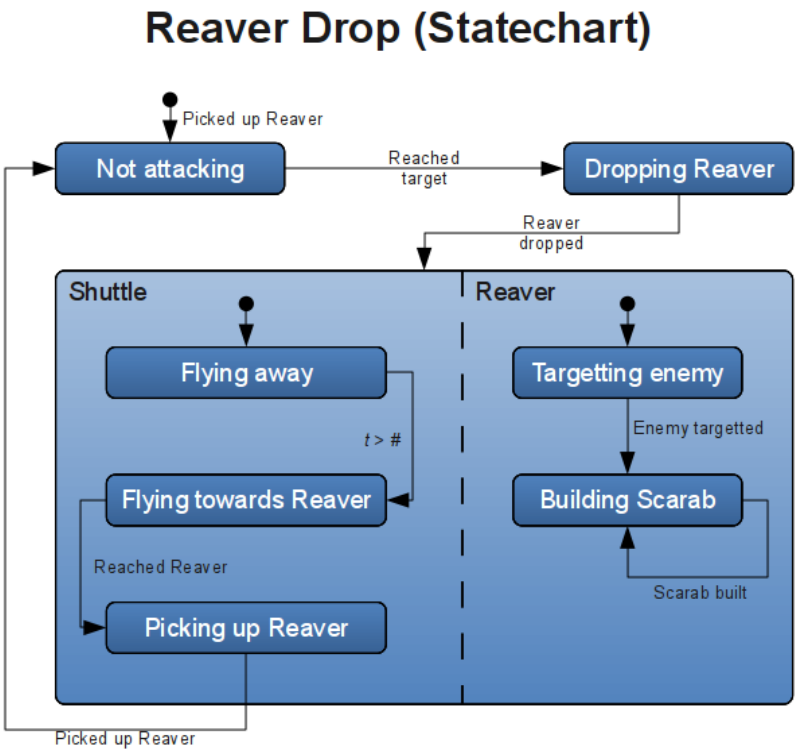
3.3.3 Actions implementation

Implements the abstract actions by managing the reactive layer and sending commands to BWAPI.

3.3.4 Reactive layer

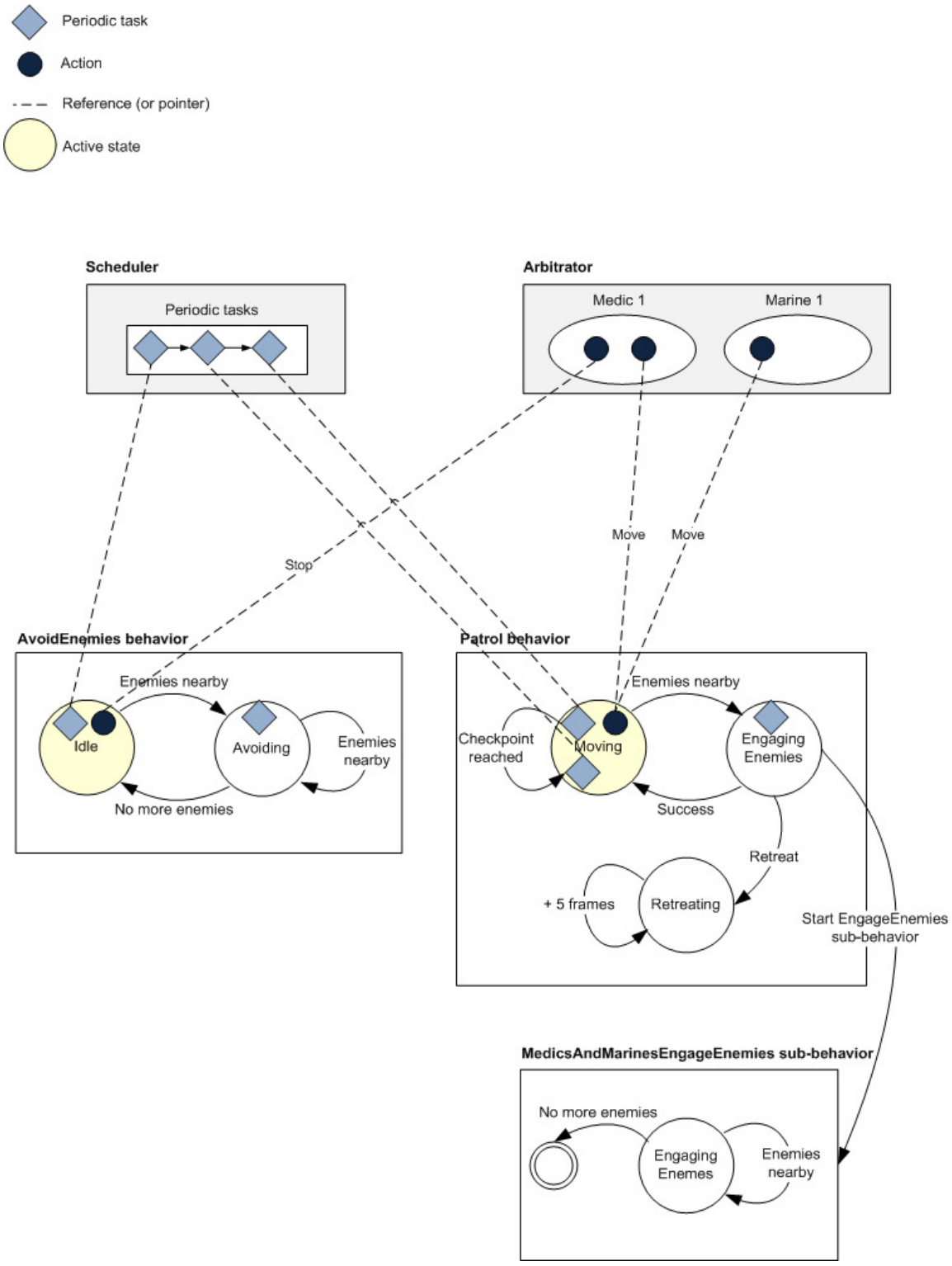
Associates reactions to environment events using Behaviours. Behaviours are represented as FSMs (Finite State Machines) - or more precisely, as statecharts. As such, they are made up of these two main things:

- States: finite number of states the behaviour may currently be in
- Transitions: links between pairs of states



The figure above depicts an example of statechart – in this case, a Reaver inside a Shuttle sent to harass the enemy. This behaviour is coded (more or less faithfully) in the SPAR agent (SparAIModule/DecisionMaking/Layer1Reactive/FSM/Behaviors/ReaverDrop).

States may contain arbitrary data. Data shared by multiple states should be put in the enclosing Behavior class.
On an abstract level, time should be considered to flow in the states, not in the transitions. When entering a new state (even if it is the same one as before), all state-based data should be re-initialized. Moreover, all sub-behaviours started in a given state are automatically terminated.



The figure above details two behaviors: AvoidEnemies and Patrol running concurrently. In this example, a Medic executes both behaviors while the Marine has only the Patrol behaviour. In this case, if two actions conflict (the Medic moving back to avoid an enemy but moving forward to patrol to the next checkpoint) the highest priority behaviour gets to have its proposed action executed.

Except for the initial behaviour startup, code execution is triggered by an event being fired. Events can be fired from SparAIModule directly (one of the callbacks) or a periodic event can be fired every x frames from the scheduler. Events are stored in BehaviorState::m_events. Transitions are stored in BehaviorState::m_transitions. Events for a particular state are automatically registered when that state is entered, and are automatically unregistered when that state is exited.
The call stack for the firing of a periodic event is described here:

-> Scheduler::onFrame() -> SchedulerTask::run() -> [...] -> Transition::onEvent() -> Transition:testTransition() [-> State::onEnter()]

The [...] is where implementation may differ. Designers may choose to:

- Have the transition subscribe directly to the event, in which case the transition's condition is evaluated right when the event is fired (in this case [...] is empty).
- Have the BehaviorState subscribe to the event instead, and call Transition::testTransition() manually. This allows more flexibility (arbitrary code) but is somewhat limiting the reuse of behaviours for plan recognition purposes.

3.4 Flow

1) Dll.cpp - The agent is instantiated in the newAIModule() function.
    - Depending on whether the current map is a scenario, the appropriate implementation of the various modules is instantiated.
2) SparAIModule.h - The agent is initialized via the SparAIModule::onStart() function.
    - BWTA analyzes the map.
3) SparAIModule.h - The agent is notified via the various callbacks when something happens. In particular, the onFrame() callback allows the agent to execute code during each frame. However, it must not exceed the allotted time, else the agent would slow down the game.
    - The events are dispatched to the appropriate components.

3.5 External components

3.5.1 Spar Logger

3.5.2 Spar GUI

4 Scenarios

The scenarios are meant to test the individual behaviour of some component (as opposed to the global behaviour of the agent). All scenarios-dependent code should go in SparAIModule/Scenarios.

4.1 Scenario 1

1 Medic vs Zerglings

4.2 Scenario 2

3 Dragoons vs 3 Dragoons

4.3 Scenario reavers

Reavers vs a lot of stuff

5 Future work

5.1 Graphical user interface

Debugging the SPAR can be done through the logging component and the introduction of breakpoints in the code (very useful since only one thread is used at the moment). However, a graphical user interface could allow one to inspect the current state of the game more easily, in particular by gaining access to the list of currently active actions and behaviours, and could also allow users to input goals to satisfy and actions or behaviours to execute. The SparQtGUI project can act as a starting point.

For an example, see http://www.youtube.com/watch?v=_41Eew_DamA illustrating the debugging capabilities of the EISBot.

5.2 Strategy inference

The tech tree dependencies can be exploited to reason about what types of units or techs the opponent has access to. Likewise, keeping count of the number of enemy units created/killed along with the amount of minerals left at the opponent's base locations can provide an estimate on the number of resources that are available to him.

5.2.1 Plan recognition

5.3 Pathfinding

At the moment, the SPAR agent relies on StarCraft built-in pathfinding to move units around the map. A better pathfinding implementation would use FSMs to closely monitor units as they move and take care of the following aspects: threat awareness and spatial reasoning.

- Section 2 (beginning at p. 60) of AI Game Programming Wisdom volume 4.
- Hagelbäck-Johansson – *A Multiagent Potential Field-Based Bot for Real-Time Strategy Games*

5.3.1 Threat awareness

Using the influence map, avoid crossing dangerous areas controlled by enemy units.

5.3.2 Spatial reasoning

Detect paths potentially not available due to neutral structures/mineral patches/player buildings blocking the way.

Articles:

- Forbus-etal – *How qualitative spatial reasoning can improve strategy game AIs*

5.4 Planning

Instead of a semi-dynamic script, the actions supplied by the layer 3 of decision making should be generated by an adversarial planning process.

Articles:

- Weber-etal – *Applying Goal-Driven Autonomy to StarCraft*

5.4.1 Resources planning

Articles:

- Chan etal – *Online Planning for Resource Production in Real-Time Strategy Games*

5.4.2 Combat planning

Articles:

- Balla-Fern – *UCT for Tactical Assault Planning in Real-Time Strategy Games*
- Sailer-etal – *Adversarial Planning Through Strategy Simulation*
- Kovarsky-Buro – *Heuristic Search Applied to Abstract Combat Games*

5.5 Learning

?