Interfacing Agents to Real-Time Strategy Games

Jensen, Andreas Schmidt; Kaysø-Rørdam, Christian; Villadsen, Jørgen

Published in:
Proceedings of the 13th Scandinavian Conference on Artificial Intelligence (SCAI 2015)

Publication date:
2015

Document Version
Peer reviewed version

Thirteenth Scandinavian Conference on Artificial Intelligence – SCAI 2015
S. Nowaczyk (Ed.) pp. 68–77
IOS Press, 2015

Interfacing Agents to Real-Time Strategy Games

Andreas Schmidt JENSEN, Christian KAYSØ-RØRDAM and Jørgen VILLADSEN [1]
Technical University of Denmark, Kgs. Lyngby, Denmark

Abstract. In real-time strategy games players make decisions and control their units *simultaneously*. Players are required to make decisions under time pressure and should be able to control multiple units at once in order to be successful. We present the design and implementation of a multi-agent interface for the real-time strategy game StarCraft: Brood War. This makes it possible to build agents that control each of the units in a game. We make use of the *Environment Interface Standard*, thus enabling different agent programming languages to use our interface, and we show how agents can control the units in the game in the *Jason* and GOAL agent programming languages.

Keywords. multi-agent systems, environment interface standard, real-time strategy games, agent programming languages

1. Introduction

We present the design and implementation of a multi-agent interface for the game StarCraft: Brood War (SCBW). SCBW is developed by Blizzard Entertainment and was released in 1998. It is a real-time strategy (RTS) game in which the player can control a number of units with different capabilities, and must use these to build an army to eliminate the enemy players. The overall goal of this paper is to provide an interface that lets each of the player's units be controlled by an intelligent agent. Due to the fast-paced nature of real-time strategy games, players are required to make decisions under time pressure and should be able to control multiple units at once in order to be successful. These requirements makes multi-agent systems (MAS) a natural choice for implementing intelligence in the game, since the agent paradigm enables each unit to make their own rational decisions. Moreover, SCBW comes with an advanced map editor, which makes it possible to create advanced scenarios for testing e.g. algorithms for cooperation and coordination.

SCBW does not include an API for interacting with the game, so the integration is done using the Brood War Application Programming Interace (BWAPI) [1], which is an unofficial framework that can interact with SCBW. The interface is widely used for AI competitions (The StarCraft AI Competition at AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment [2]; IEEE Conference on Computational Intelligence and Games [3]; Student StarCraft AI Tournament [4]; StarCraft Micro AI

[1] Corresponding Author: Jørgen Villadsen, DTU Compute, Richard Petersens Plads, Building 324, 2800 Kgs. Lyngby, Denmark; E-mail: jovi@dtu.dk.

Tournament [5]), and enabling dedicated agent programming languages (APLs) to interact with SCBW makes it easier for MAS researchers to use these competitions to test the capabilities of their agents.

The Environment Interface Standard (EIS) "facilitates connecting agents programmed in various agent platforms to environments" [6]. It is designed to be generic and to be easily integrated into existing agent platforms. An agent in EIS is very minimalistic: it can perform actions and perceive from the environment. This makes it possible to use EIS in many different APLs, since most platforms already support the notions of actions and percepts. The list of APLs supporting EIS includes *Jason* [7], GOAL [8], 2APL [9], and Jadex [10]

Our contribution is thus an interface to the BWAPI, implemented using EIS, such that APLs adopting EIS can readily and easily connect agents to SCBW. This allows MAS researchers to e.g. test their agents against succesful implementations of SCBW AIs, or to build scenarios (using the map editor), which can illustrate specific aspects of the agents' capabilities. The implementation is open source and available at `http://people.compute.dtu.dk/ascje/IAiG/`.

The rest of the paper is organized as follows. In section 2, we briefly describe SCBW and the API that makes the integration between the game and multi-agent systems possible. We describe how the integration is done in section 3 and show how agents can interact with the environment in different APLs in section 4. We conclude the paper in section 5.

2. Real-Time Strategy Games

A real-time strategy game is a strategic video game in which players make decisions and control their units *simultaneously*. This is in contrast with turn-based strategy games in which the players take turn, and the game only progresses when a player has made a move. Chess is probably the most well-known turn-based strategy game, however in the video game genre, games such as *Civilization* and *Heroes of Might and Magic* are quite popular.

2.1. StarCraft: Brood War

STARCRAFT: BROOD WAR is a real-time strategy game in which a player controls one of three distinct races. The three races, Terran, Zerg and Protoss, consist of a number of unique units, making the tactics required for succeeding with each race quite different. A game in SCBW is usually played as a match between a number of teams trying to eliminate each other. Each team initially has a base with four workers and a main building for constructing additional workers. The workers can collect resources (minerals and gas), and use these to construct a larger base with a strong defense and an army of offensive units. The game is won, once the player has eliminated all of the oppponents.

Compared to turn-based strategy games, real-time strategy games requires players to make decisions under time pressure and they should be able to control multiple units at once in order to be successful. For example, by using resources to produce additional workers, the player's income will increase, but this does not directly increase the offensive or defensive capabilities, so if an opponent decides to build offensive units instead,

the player is left vulnerable. However, since the player's income is now higher than the opponent's, he can produce offensive units at a higher rate than the opponent, which makes it possible to build a strong defense, so if it takes too long for the opponent to mobilize his force, the player may still be successful. This is the core of the gameplay of SCBW (and RTS games in general): balancing economy with military strength. Furthermore, by knowing the strengths and weaknesses of different units, it is possible to produce a more successful army (e.g. by predicting the opponent's choice of units and making decisions based on this).

In a battle, decisions have to be made in real-time and a wrong decision could mean that the player loses. Decisions such as where each unit should position itself to maximize its efficiency, or which group of enemies to attack to gain the upper hand have to be made all the time. A human player is not able to keep up with all the decisions that have to be made, and is forced to act on a more macroscopic level, controlling large groups of units at the same time rather than individuals. By controlling each individual unit using a dedicated agent, micromanagement becomes quite natural, since the agents are able to react to local changes. Furthermore, the social aspect of agents lets them communicate and coordinate their actions, enabling them to make sound decisions quickly. For example, by using the knowledge of the opponent units' capabilities, the agents can decide amongst themselves which of the opponent units to attack.

2.2. The Brood War API

Since SCBW does not provide an API for controlling the player's units, we use BWAPI, a free and open source C++ framework that is able to read the game state using so-called DLL injection. The API provides classes for retrieving this information and for issuing commands to the game. The API is well-documented and makes it possible to build quite successful AIs for SCBW. Our purpose is, however, to use the API in APLs using EIS, and most of the well-known APLs supporting EIS are implemented in Java. On top of BWAPI, we therefore use JNI-BWAPI [11], a Java-based interface for BWAPI. JNI-BWAPI enables Java programmers to read the game state and issue commands, and is therefore useful for our integration with EIS.

The API offers most of its functionality in the `Game` class. This includes retrieving visible units, information about the map and information about each player. Units are represented by a `Unit` class, which is used to issue orders to the specific unit (move, attack, harvest resources, etc.) and to read the state of the unit (location, health points, its current task, etc.). For example, we can issue a command for making a unit move as follows:

```
api.getUnit(unitId).move(new Position(x, y), false);
```

This indicates that the unit with id `unitId` should move to the coordinate (x,y). The second parameter, `false`, indicates that it should be executed immediately. The `Map` class offers general information about the size of the map, and furthermore possible locations for building a base and for setting up a defense perimeter around such base. Finally, the `Player` class contains information about available resources, use of supplies, whether a player is an ally or an enemy, and so on.

As we can see, the API offers a lot of useful functionality, but in a centralized manner. Even though some information is conveniently put inside the `Unit` class, other infor-

Table 1. The table shows some of the percepts available to the units. All available percepts can be found on the project's website.

Percept	Description	Perceived by
gameStart	The game has started.	All
id(*Id*)	The unit is identified by *Id*.	All
position(X,Y)	The unit is located at (X,Y).	All
gathering(*Type*)	The unit is gathering resources of the given type.	Workers
minerals(*Quantity*)	The amount of minerals available.	Buildings, Workers

mation (such as available resources or the location of enemy bases) is stored elsewhere. The implementation of the interface should thus ensure that the relevant information is provided to each agent.

2.3. Related work

The BWAPI has been used in several papers on implementing AI for RTS[2]. This paper differs from most of them in that we do not present a novel way to implement AI for RTS, but rather provide a tool that allows people in the agent community to do so. While most of the work has been using techniques such as probabilistic models, neural networks and genetic algorithms, some of the work is based on the agent-paradigm.

In [12] the Nova bot, which uses a real-time multi-agent architecture with a focus on efficient communication, is presented. The system uses agents for both micro- and macro-management. Another bot, the KhasBot [13], uses the Java Agent Development Framework (JADE) [14], in which the managing of different parts of the system is delegated to different agents. This is a common trait for most of the implemented AIs; that management is performed at a relatively high level, which means that most of the reasoning is performed by agents that control more than one unit.

While our implementation lets each unit be controlled by an agent, it is also possible to create macro-managing agents that, e.g., take care of resource management or combat tactics. For example, a resource manager could control resource usage: a worker trying to construct a building should request resources from the manager before constructing it.

3. Integration – Percepts and Actions

In this section, we describe how we use the information from the API to create percepts and offer actions to the agents connecting via EIS. Our implementation interacts with SCBW through a *listener* that is informed when certain events occur, such as the start of a new game or creation or destruction of a new unit. It also provides a method for letting units execute actions.

An EIS-enabled environment is implemented by implementing the EIS interface, which requires us to provide methods for letting an agent perceive the environment and execute actions.

Perceiving the environment The kinds of percepts a unit can perceive depends on the type of unit. For example, only units that are able to *use* resources are able to perceive the amount of resources available. In general, units can perceive their name, position,

[2]See `https://github.com/bwapi/bwapi/wiki/Academics` for a comprehensive list of papers.

Table 2. The table shows some of the actions available to the units. All available actions can be found on the project's website.

Action	Description	Available to
move(X,Y)	The unit moves to (X,Y).	All
attack(*Target*)	The unit attacks the unit identified by *Target*.	Offensive units
build(*Type*,X,Y)	The unit builds a building of type *Type* at (X,Y).	Workers
gather(*Id*)	The unit begins gathering the resources identified by *Id*.	Workers

information about the map, and the location of friendly and known enemy units. Workers can perceive the amount of available resources, since they use this knowledge to decide what to build. Units with special abilities are able to perceive when these abilities are in effect. Some of the available percepts are shown in table 1.

Executing actions When an action is executed in an EIS environment, it is checked whether the action is valid and can be executed by the given agent. In SCBW, this depends on the type of the unit trying to execute the action. For example, executing an action to gather minerals can only be done by workers, and use of special abilities can only be done by units with those abilities. In our integration, an action performed by an agent is put into a queue of *pending actions*. In the BWAPI listener, this list is processed when BWAPI is ready to execute actions. An agent can only have *one* action in the queue at a time, but since the list is processed by BWAPI several times per second, this is generally not an issue. Some of the available actions are shown in table 2.

4. Examples

The implementation of the BWAPI in EIS makes it possible to program agents for SCBW in different APLs. In this section, we show that such agents can indeed be implemented in both *Jason* [7] and GOAL [8]. We assume that the reader is familiar with the APLs, so we do not go into details with the specifics of the languages. While the example is simple, it illustrates the possibilities of using SCBW as an agent environment.

4.1. Jason

A MAS in *Jason* is defined in a `mas2j` file, where the environment and agents are defined. We therefore define each of the agent types here:

```
MAS scbw {
  infrastructure: Centralised
  environment: jason.eis.EISAdapter("EISBW.jar")
  agents: terranSCV; terranCommandCenter; ...
}
```

A program for the Terran worker unit, the Space Construction Vehicle (SCV), is shown in listing 1. The first part of the program consists of belief rules, which the agents use to determine if they should construct a building. The agent can construct supply depots if the number of available supplies is low, and barracks, once the first supply depot has been built. The `canBuild` determines if the agent can construct a specific building. The agents use the position of other SCVs to determine who should construct

Listing 1: *Jason* program for the Terran SCV.

```
/* Belief rules */

cost("Terran Supply Depot", 100, 0).
cost("Terran Barracks", 150, 0).

condition("Terran Supply Depot") :- supply(C, Max) & Max - C < 6.
condition("Terran Barracks") :- supply(_, Max) & Max > 20 &
  not(friendly(_, "Terran Barracks", _, _, _)).

canBuild(Building, X, Y) :- condition(Building) & cost(Building, M, G) &
  minerals(MQ) & M <= MQ & gas(GQ) & G <= GQ &
  friendly(_, "Terran Command Center", Id, _, _) &
  jia.findBuildingLocation(Id, Building, X, Y) &
  position(MyX,MyY) &
  jia.tileDistance(MyX,MyY,X,Y,D) &
  .findall(_, (friendly(_, "Terran SCV", _, OtherX, OtherY) &
        jia.tileDistance(OtherX,OtherY,X,Y,OtherD) &
        OtherD < D), []).

/* Plans */

+gameStart <- !work.

+!work : canBuild(Building, X, Y)
  <- !build(Building, X, Y); .wait(1000); !!work.
+!work : not(gathering(_)) & mineralField(Id, _, _)
  <- gather(Id); .wait(1000); !!work.
+!work <- .wait(200); !work.

+!build(Building, X, Y) : cost(Building, M, G) &
    minerals(MQ) & M <= MQ & gas(GQ) & G <= GQ
  <- build(Building, X, Y).
+!build(Building, X, Y) <- .wait(200); !build(Building, X, Y).
```

the building. We use *internal actions* to find a proper location close to the command center. Internal actions in *Jason* makes it possible to write Java code that can interact directly with the API to, e.g., compute locations for constructing a building. The SCV closest to the location will then begin constructing the building.

The second part of the program consists of plans. The first plan lets the agents react to the fact that the game has started. They adopt the goal `!work`, which they keep in order to continuously gather resources and construct buildings. The first `!work` plan is applicable if the agent can construct a building. Otherwise, if the agent is not collecting resources, it will start collecting minerals. The agent then continuously checks if something should be done (i.e., if a building should be constructed). Finally, the last plans are used for actually constructing buildings. Note that we check for sufficient resources both in the `canBuild` rule and in the plan context, since the resources may have been used by another agent. If that is the case, the agent waits and then adopts the goal again.

4.2. GOAL

A MAS in GOAL is defined in a `mas2g` file, where the environment and agents are defined. Furthermore, a launchpolicy is defined, which specifies rules for when to create the agents. In our case, we simply launch an agent designed for the type of unit, e.g. a `terranSCV` agent when a Terran SCV is spawned in the game.

```
environment {
  env = "EISBW.jar".
}

agentfiles {
  "TerranSCV.goal" [name = terranSCV].
  "TerranCommandCenter.goal" [name = terranCommandCenter].
  ...
}

launchpolicy {
  when [type = terranSCV]@env do launch terranSCV:terranSCV.
  when [type = terranCommandCenter]@env
       do launch terranCommandCenter:terranCommandCenter.
  ...
}
```

An implementation of the Terran SCV in GOAL is shown in listing 2 and continued in listing 3. GOAL differs from *Jason* in that we cannot gain access to BWAPI. In order to find viable locations for construction, we therefore generate percepts (`constructionSite(X,Y)`) that can be used for finding the most viable location. The main program checks if the agent can construct a building (based on the same conditions as the *Jason* program), and if not, the agent starts gathering minerals.

An implementation of the Terran Command Center in GOAL is shown in listing 4.

5. Conclusion

We have presented an interface between SCBW and various APLs using EIS. We furthermore described implementations of an agent controlling worker units in both *Jason* and GOAL. The implementations show that our EIS-enabled interface to SCBW makes it possible to build agents controlling the units in real-time strategy games. By using EIS, we hope that others will take part in implementing AI for SCBW using intelligent agents. The interface is therefore open source and is available online.

We furthermore believe that one of the major strengths of using SCBW is the fact that it, as mentioned, comes with a map editor, allowing researchers to make advanced scenarios for testing specific multi-agent situations. Since our focus for this paper has been to construct a useful interface, we have not looked into this yet, but we plan to do so in the future.

Listing 2: GOAL program for the Terran SCV.

```
init module {
  beliefs {
    busy :- constructing ; gathering(_).
    cost("Terran Supply Depot", 100, 0).
    cost("Terran Barracks", 150, 0).
    condition("Terran Supply Depot") :-
      supply(C, Max), Max - C < 6.
    condition("Terran Barracks") :-
      aggregate_all(count,friendly(_, "Terran Barracks", _, _, _),0),
      supply(_, Max), Max > 20.
    canBuild(Building, X, Y) :-
      condition(Building), cost(Building, M, G),
      minerals(MQ), M =< MQ, gas(GQ), G =< GQ,
      friendly(_, "Terran Command Center", _, TX, TY),
      buildingLocation(TX, TY, X, Y),
      position(MyX,MyY), distance(MyX,MyY,X,Y,D),
      findall(_, (friendly(_, "Terran SCV", _, OtherX, OtherY),
        distance(OtherX,OtherY,X,Y,OtherD), OtherD < D), []).
    buildingLocation(X,Y, RX,RY) :-
      findall([D,BX,BY],
        (constructionSite(BX,BY), distance(X,Y,BX,BY,D)), L),
      sort(L, [[_,RX,RY]|_]).
    distance(X1,Y1,X2,Y2,D) :- D is sqrt((X2-X1)**2 + (Y2-Y1)**2).
  }
  program {
    if bel(percept(id(Id))) then insert(id(Id)).
    if bel(percept(supply(C,M))) then insert(supply(C,M)).
    if bel(percept(minerals(M))) then insert(minerals(M)).
    if bel(percept(gas(G))) then insert(gas(G)).
    if bel(percept(position(X,Y))) then insert(position(X,Y)).
  }
}

main module{
  program {
    if bel(not(constructing), canBuild(Building, X, Y))
      then insert(constructing) + build(Building, X, Y).
    if bel(not(busy), mineralField(Id)) then gather(Id).
  }
  actionspec {
    gather(Id) {
      pre { not(gathering(X)) }
      post { true }
    }
    build(Building, X, Y) {
      pre { cost(Building, M, G), minerals(MQ), M =< MQ,
        gas(GQ), G =< GQ }
      post { true }
    }
  }
}
```

Listing 3: GOAL program for the Terran SCV (continued).

```
event module {
  program {
    forall bel(not(percept(supply(C,Max))), supply(C, Max))
      do delete(supply(C,Max)).
    if bel(percept(supply(C,Max)), not(supply(C, Max))) then
      insert(supply(C,Max)).

    if bel(percept(gathering(X)), not(gathering(X)))
      then insert(gathering(X)).
    if bel(gathering(X), not(percept(gathering(X))))
      then delete(gathering(X)).

    if bel(not(percept(minerals(M))), minerals(M))
      then delete(minerals(M)).
    if bel(percept(minerals(M)), not(minerals(M)))
      then insert(minerals(M)).
    if bel(not(percept(gas(G))), gas(G)) then delete(gas(G)).
    if bel(percept(gas(G)), not(gas(G))) then insert(gas(G)).

    forall bel(percept(mineralField(Id,_,_,_,_)), not(mineralField(Id)))
      do insert(mineralField(Id)).

    forall bel(percept(id(MyId)), percept(friendly(Name,Type,Id,X,Y)),
      MyId \= Id, not(friendly(Name,Type,Id,X,Y)))
      do insert(friendly(Name,Type,Id,X,Y)).

    if bel(percept(position(X1,Y1)), position(X2,Y2),
      (X1 \= X2 ; Y1 \= Y2))
      then insert(position(X1,Y1)) + delete(position(X2,Y2)).

    forall bel(percept(constructionSite(X,Y)),
      not(constructionSite(X,Y)))
      do insert(constructionSite(X,Y)).
    forall bel(not(percept(constructionSite(X,Y))),
      constructionSite(X,Y))
      do delete(constructionSite(X,Y)).

    if bel(percept(constructing), not(constructing))
      then insert(constructing).
    if bel(not(percept(constructing)), constructing)
      then delete(constructing).
  }
}
```

Listing 4: GOAL program for the Terran Command Center.

```
main module{
  beliefs{
    trainSCV.
  }
  program{
    if bel(aggregate_all(count,(percept(
      friendly(_, "Terran SCV", _, _, _))),X),X<25)
      then train("Terran SCV").
  }
  actionspec {
    train(Id) {
      pre {trainSCV}
      post {not(trainSCV)}
    }
  }
}

event module {
  program {
    if bel(not(trainSCV), percept(queueSize(N)), N < 2,
      percept(minerals(X)),X>=50,percept(supply(Y,Z)),Y<Z)
      then insert(trainSCV).
  }
}
```

References

[1] BWAPI: An API for interacting with Starcraft: Broodwar. http://bwapi.github.io/, 2015.

[2] AIIDE StarCraft AI Competition. http://www.starcraftaicompetition.com/, 2015.

[3] CIG StarCraft AI Competition. http://cilab.sejong.ac.kr/sc_competition/, 2015.

[4] Student StarCraft AI Tournament. http://sscaitournament.com/, 2015.

[5] SC Micro AI Tournament. http://scmai.hackcraft.sk/, 2015.

[6] Tristan M. Behrens, Koen V. Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61(4):261–295, 2011.

[7] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[8] Koen V. Hindriks. Programming Rational Agents in GOAL. *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157, 2009.

[9] Mehdi Dastani. 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, June 2008.

[10] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In *Multi-Agent Programming*, pages 149–174. 2005.

[11] jnibwapi – JNI interface for BWAPI. https://code.google.com/p/jnibwapi/, 2015.

[12] Alberto Uriarte. Multi-reactive planning for real-time strategy games. Master's thesis, Universitat Autònoma de Barcelona, 2011.

[13] Antonio Arredondo, Daniel Jaramillo, Frank Natividad, and Ben Wright. Agent-Oriented Programming Framework Design for Real-Time Strategy AI. 2011.

[14] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, 2007.