Designing a Cognitive Agent Connector for Complex Environments
A Case Study with StarCraft

Koeman, Vincent J.; Griffioen, Harm J.; Plenge, Danny C.; Hindriks, Koen V.

DOI
10.1007/978-3-030-25693-7_16

Publication date
2019

Document Version
Final published version

Published in
Engineering Multi-Agent Systems

Designing a Cognitive Agent Connector for Complex Environments: A Case Study with StarCraft

Vincent J. Koeman(✉), Harm J. Griffioen, Danny C. Plenge, and Koen V. Hindriks

Delft University of Technology, Delft, The Netherlands
{v.j.koeman,h.j.griffioen,p.c.plenge,k.v.hindriks}@tudelft.nl

Abstract. The evaluation of cognitive agent systems, which have been advocated as the next generation model for engineering complex, distributed systems, requires more benchmark environments that offer more features and involve controlling more units. One issue that needs to be addressed time and again is how to create a connector for interfacing cognitive agents with such richer environments. Cognitive agents use knowledge technologies for representing state, their actions and percepts, and for deciding what to do next. Issues such as choosing the right level of abstraction for percepts and action synchronization make it a challenge to design a cognitive agent connector for more complex environments. The leading principle for our design approach to connectors for cognitive agents is that each unit that can be controlled in an environment is mapped onto a single agent. We design a connector for the real-time strategy (RTS) game StarCraft and use it as a case study for establishing a design method for developing connectors for environments. StarCraft is particularly suitable to this end, as AI for an RTS game such as StarCraft requires the design of complicated strategies for coordinating hundreds of units that need to solve a range of challenges including handling both short-term as well as long-term goals. We draw several lessons from how our design evolved and from the use of our connector by over 500 students in two years. Our connector is the first implementation that provides full access for cognitive agents to StarCraft: Brood War.

1 Introduction

Multi-agent systems, consisting of multiple autonomous agents interacting with an external environment, have been promoted as the approach for handling problems that require multiple problem solving methods, multiple perspectives, and/or multiple problem solving entities [8]. In the past twenty years, the research community has combined multi-agent system (MAS) concepts and approaches into mature frameworks for agent-oriented programming (AOP) [2,15]. Current cognitive agent technology thus offers a viable and promising

An earlier version of this work was presented at the 2018 EMAS workshop [10].

© Springer Nature Switzerland AG 2019
D. Weyns et al. (Eds.): EMAS 2018 Workshops, LNAI 11375, pp. 302–319, 2019.
https://doi.org/10.1007/978-3-030-25693-7_16

alternative to other approaches for engineering complex distributed systems [6,14]. However, Hindriks [6] also concludes that "if [cognitive] agents are advocated as the next generation model for engineering complex, distributed systems, we should be able to demonstrate the added value of [multi] agent systems."

Designing a connector that can demonstrate this added value by connecting cognitive agents with an environment that puts strict real-time constraints on the responsiveness of agents, requires coordination at different levels (ranging from a few agents to large groups of agents), and requires complex reasoning about long-term goals under a high level of uncertainty is not a trivial task. The connectors that are currently available for use with cognitive agent systems have remained rather simple, and thus do not fully demonstrate the added value of cognitive agent technology.

In this chapter, we aim to establish a *design approach for developing connectors for complex environments*, aimed at facilitating the development of more connectors that can be used to demonstrate the ease of use of cognitive technologies for engineering large-scale complex distributed systems for challenging environments. We believe that RTS games that deploy large numbers of units provide an ideal case study to this end [4,17]. The basic idea is to control each unit with a cognitive agent. Based on this, and in accordance with Google (DeepMind) and many other AI researchers [13,16], we believe that StarCraft is the most suitable RTS game to target in our case study. Moreover, several popular competitions exist for StarCraft AI that can serve as a benchmark for implementations that use cognitive technologies [16]. By carefully designing and efficiently implementing a cognitive agent connector to StarCraft, and then testing this connector with large groups of students, we iteratively refine our approach for the development of agent-environment connectors.

Our focus in this work is on the case study of designing a connector that enables and facilitates the use of cognitive agent technology for engineering strategies for StarCraft (Brood War) based on a *one-to-one unit-agent mapping*, which is different from most existing StarCraft AI implementations. This unit-agent mapping introduces important challenges that need to be addressed:

1. The connector should facilitate a MAS that operates at a level of *abstraction* that is appropriate to cognitive agents.
2. The connector should be sufficiently *performant* in order to support a sufficient variety of viable MAS implementations using cognitive agents (i.e., both different approaches to implementing strategies as well as the use of different agent platforms).

In other words, the connector design should not force a cognitive MAS to operate at the same level of detail as bots written for StarCraft in C++ or Java, but also not promote the other extreme and abstract too much (e.g., clearly the extreme abstraction of providing a single action 'win' is not useful). To make optimal use of the reasoning typically employed by cognitive agents, the connector should leave low-level details to other control layers whilst still allowing agents sufficiently fine grained control.

The remainder of this chapter is organized as follows. In Sect. 2, we discuss the current state-of-the-art in environments available for cognitive agents. Next, in Sect. 3, we introduce StarCraft as a case study for connector design. In Sect. 4, we detail our design approach of a multi-agent connector by introducing general guidelines, applying them to our case study, and discussing the lessons learned from this. Finally, Sect. 5 concludes this chapter with recommendations for future work on both cognitive agent connectors as well as cognitive agent technologies in general.

2 Related Work

Connectors that support connecting cognitive agent technology to games have been made available for other games [3]. So far, however, most connectors have remained rather simple. The most complex cognitive multi-agent connectors that have been made available so far, are connectors for Unreal Tournament [7]. The design of such a connector involves similar issues related to the facilitated level of abstraction and the resulting performance as in this work. However, the resulting implementation as reported on by Hindriks [7] does not support running more than 10 agents, whereas for a StarCraft interface we need to connect hundreds of cognitive agents to control the hundreds of units in game. Moreover, corresponding agent systems for Unreal Tournament generally offer only a very restricted set of actions that agents can perform (i.e., mostly just a "go to" action because other middleware software is used to take care of path planning, shooting, etc.) or communication (i.e., mostly just informing others about enemy positions), limiting the complexity of decision making that is required. Relatively speaking, compared to StarCraft, the diversity in strategies or tactics that can be deployed is rather small. Another problem related to Unreal Tournament is that games cannot be sped up, complicating testing and debugging. It is therefore not feasible to derive a design approach for connectors to richer environments from this work.

RTS games are widely regarded as an ideal testbed for AI [13,17]. An RTS game like StarCraft involves long-term high-level planning and decision making, but also short term control and decision-making with individual units. This distinction between respectively strategical and tactical decision making is generally referred to as *macro* and *micro* respectively. These factors and their real-time constraints with hidden information make RTS games like StarCraft ideal for iterative advancement in addressing fundamental AI challenges [17]. Although machine learning solutions have been applied to some problems at the micro level, learning techniques have not been successfully applied to other aspects, mainly due to the vast state spaces involved [16]. The concepts of cognitive agents seem to be a good fit for addressing these challenges, allowing individual cognitive agents to reason about their tactical decision making whilst also inherently facilitating communication to make decisions at a joint strategical level. The reasoning typically applied by cognitive agents seems to lend itself for macro really well, but such systems can potentially employ learning techniques to perform specific sub-tasks (at the micro level) as well. A cognitive agent connector can also facilitate the use of MAS as an approach for allowing several individual AI techniques to work together.

The work of Weber et al. [19] recognizes the value of agent-oriented techniques for StarCraft AI. Their "EISBot" uses a reactive planner combined with external components like case-based reasoning and machine learning. Similar to multi-agent systems, the concepts of percepts and actions are used. However, there is only a single 'agent' that is compartmentalized into several specific managers. This approach is thus still based on a single-bot approach, whilst in this work, we instead aim to design a connector for multi-agent systems in which each in-game unit is connected to an individual cognitive agent. Moreover, it is not made clear which percepts and actions are provided, and what the gain in terms of abstraction level and the loss in terms of performance in this implementation is, as the focus is on the implementation of the StarCraft bot itself, instead of on the design of a (generic) connector as in this paper.

The prototypical RTS game is StarCraft [16], originally developed by Blizzard in 1998, but still immensely popular both in (professional) gaming and AI research. An API for StarCraft (Brood War) has been developed for several years: BWAPI [5]. BWAPI reveals the visible parts of the game state to AI implementations, facilitating the development of competitive (non-cheating) bots. Several dozens of such bots have been created with this API, mostly written in C++ or Java, aimed at participating in one of the tournaments that are being held for StarCraft AI implementations. However, this work does not directly facilitate cognitive agents that use knowledge technologies and realise a one-to-one unit-agent mapping.

A first attempt at creating a cognitive interface for StarCraft was performed by Jensen et al. [9]. In this work, a working proof-of-concept that ties in-game units to cognitive agents was introduced. However, it does not address the major challenges such an implementation faces concerning the level of abstraction and corresponding performance, as we do in this work. When using this connector, it is not possible to create viable (diversities of) strategies, as the range of strategies it supports is quite limited. This connector only offers a small subset of all possible actions associated with each unit in the game, and the percepts made available by the connector do not provide sufficient information for in game decision making either. In this work, we aim to allow virtually any strategy to be implemented with a sufficient level of performance using a cognitive agent connector based on the design approach we propose.

3 Case Study: StarCraft

In StarCraft, each of the three playable races have their own set of unit types, with roughly 15 types of air/ground units and 15 types of buildings per race. Although many races share similar types of buildings (e.g., depots to bring resources to), there are also substantial differences to take into account (e.g., one race requiring units to 'morph' into a different type of unit). For most types of units, there are usually multiple 'instances' (i.e., individual units) in a game, thus allowing anywhere from 5 up to 400 units representing one army in the game at a certain time. Depending on factors such as game length, the average number of units for an army in a typical game at any point in time is around 100, although many units will also die during the game (i.e., the total number of agents used is much higher). Performance is thus of vital importance, as a substantial performance impact caused by large amounts of percepts for example, will limit the amount of viable strategies.

Our cognitive agent connector to StarCraft was developed and refined in three iterations. We draw several general lessons from these iterations, which we have incorporated into our proposal for a connector design approach. Initially, a pilot was held with around 100 Computer Science master's students that worked in groups on creating a StarCraft bot using this connector. Shortly after, over 200 first year Computer Science bachelor's students did the same with an improved version of the connector, being the largest StarCraft AI project so far. We continued development of the connector after this project, and made several additional improvements, after which 300 first year Computer Science bachelor's students used the 'final version' of our connector.

4 Connector Design Approach

In this section, we discuss our design approach for a cognitive agent connector. The core of such a connector consists of three components: (i) the *entities* that are provided for agents to connect to (i.e., units in an RTS game), (ii) the outputs that are generated by each entity (and thus which *percepts* a corresponding agent receives), and (iii) the inputs that are available for each entity (and thus which *actions* an agent controlling the entity can perform). This structure is illustrated in Fig. 1. Each of these aspects will be discussed, starting with general guidelines, their application to our case study of StarCraft, and the refinements that were made after practical use of the StarCraft connector. Next, key steps for evaluating whether the connector design is fit for use in practice for developing cognitive MAS will be given and performed for our connector.

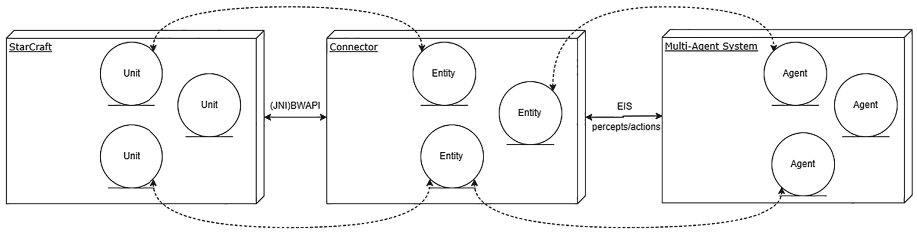


Fig. 1. An overview of the various components, with StarCraft on the left, our connector in the center, and a cognitive agent system playing the game on the right.

We make some basic assumptions about the architecture of a *cognitive agent*, as illustrated in Fig. 2. We assume such an agent pro-actively reasons about the *actions* that it should take based on (for example) its goals and beliefs in some

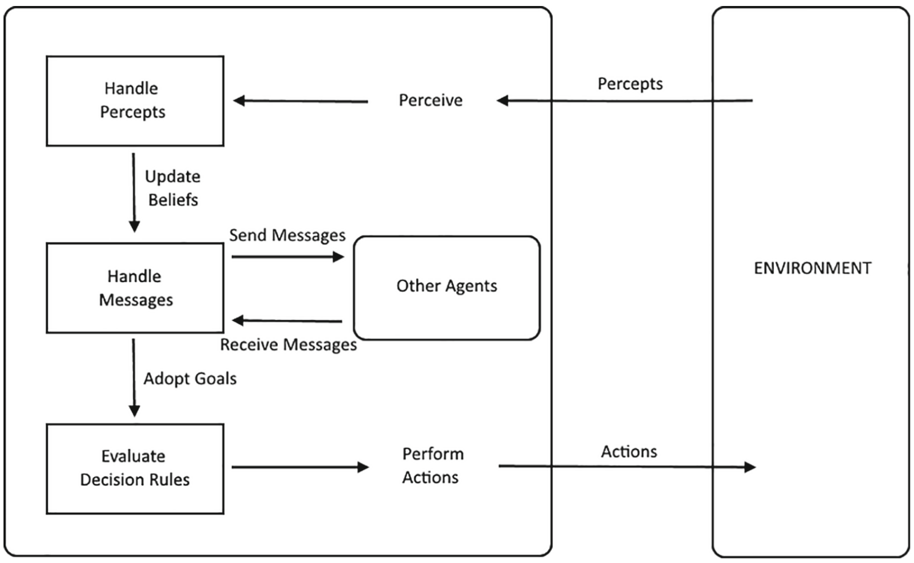


Fig. 2. The assumed structure of a cognitive agent in a multi-agent system (left) interacting with an external environment (right).

fixed *decision cycle* that is asynchronous from the environment in which it operates (for a certain *entity* in that environment), from which it receives information through *percepts*. Multiple agents can work together in one multi-agent system, which is not centrally controlled but does facilitate direct *messaging* between (groups of) agents. Our connector makes use of the Environment Interface Standard [1] in order to facilitate interacting with MAS platforms.

4.1 Micro and Macro Management

In complex environments such as StarCraft, a crucial distinction exists between top-down strategical decision making (macro) and bottom-up tactical decision making (micro). The basic assumption that we make is that a connector needs to provide support for a multi-agent approach based on a one-to-one unit-agent mapping, which inherently facilitates decision making from a bottom-up perspective. At the micro level, *every unit that is active in the environment should be mapped onto an entity* that a cognitive agent can connect to in order to control the behaviour of the unit. For StarCraft, this thus means that any moving or otherwise active unit such as a building will be controlled by a cognitive agent.

Although we initially assumed that the emergent behaviour from these agents would be sufficient to cover the strategical aspects, in practice this was hindered by the high dynamicity of an environment such as StarCraft, for example illustrated by the fact that any unit can be killed at any point in time. To facilitate macro management, we therefore have introduced a new, special kind of entities, so-called *managers*, which are made available by the connector. Managers do not match with unique in-game units, and as such they do not naturally have percepts or actions associated with them. However, as they still need to be informed about the state of the game in order to perform strategical decision making, they instead should have the ability to receive desired global information through percepts, as for example indicated by a developer in the initialization settings of a MAS.

Manager agents are especially useful to reason about groups of units. For example, without managing agents, all agents for resource gathering units in StarCraft (of which there are generally several dozen) would have to process information about the available resources and resource depots (i.e., subscribe to the relevant percepts and handle them), and then coordinate amongst each other about the division of tasks (i.e., implement some decentralized messaging protocol). Instead, a single manager agent can be the only one to have to deal with all the information about resources, and then use this information to assign a task to each resource gathering unit (i.e., through messaging), whilst in contrast the agents for those units would still handle defending themselves for example. This significantly reduces the total amount of percept processing and message sending that is required in such a situation. Moreover, in our case study we found that there is a need for dynamically adding or removing managers in order to for example temporarily centralize the reasoning for a group of attacking units, which is another frequently occurring situation in which using managers is beneficial for both performance and the effectiveness of the coordination between the relevant agents. The specific type and choice of managers that are made available by a connector and the resulting organizational structure is, however, not specified in our design approach so as to facilitate as many multi-agent system structures as possible. As there is information that is specific to certain units (and thus specific agents), and each unit has its own set of actions (which a single agent needs to call), it is not possible to completely centralize the reasoning.

Because our approach is to provide an entity (i.e., to which an agent can connect) for each unit, and the available actions for each unit are mainly defined by the (interface to) the environment itself, the main challenge when balancing the level of abstraction with the resulting performance is in determining the percepts that are available. As we assume cognitive agents here that explicitly represent their beliefs and goals, this essentially means we need to design an ontology that includes all relevant concepts for representing and reasoning about the environment at an appropriate level of abstraction.

4.2 Local and Global Information

The set of available percepts determines what information a specific entity 'sees' during the game, and thus what information its corresponding agent will receive. Percepts have a *name* to describe them and a set of *arguments* that contain the actual data. For example, a percept could be defined as `map(Width, Height)`, and an agent could then receive `map(96, 128)` in a match. In order to determine the percepts that are created for each type of unit, our approach proposes several design guidelines. A key foundation of our approach to handling information from complex environments such as StarCraft is that there is a difference between '*local*' information that is specific to a certain unit in the game (e.g., a unit's health) and '*global*' information that is potentially relevant to all units (e.g., the locations of enemy units). An agent should be able to *perceive all local information* that is specific to its corresponding unit's state, whilst a manager agent should be able to *perceive all global information* that is needed for its strategic (macro) reasoning. However, pieces of global information might also be needed in the agent for a specific unit (e.g., nearby enemy units in StarCraft).

To this end, we initially pushed all global information to all unit and manager entities, as a connector cannot determine which parts of this information a specific agent will need. However, our case study showed that this caused a significant performance impact with larger numbers of units. We have therefore found it useful to provide specific mechanisms to a developer to fine-tune the delivery of global percepts. Through the connector's initialization settings, a list of desired 'global information' (i.e., names of percepts) can be given ("subscribed to") for each unit type. For example, the (pseudocode) initialization rule `zergHatchery: [friendly, enemy]` will ensure that all agents for all *Zerg Hatchery* entities in a match will receive information about all friendly units and all visible enemy units. In this way, a developer can decide which information is relevant for certain agents, instead of such information being sent to agents at all times. This mechanism can also be used for specifying in more detail which global information a certain manager agent needs to be made aware of. Finally, we assume that when local information is needed for macro reasoning, this can be sent to the appropriate manager agent by the agent for a specific unit within the agent platform; it is thus not required to handle this within the connector (design) itself, as illustrated by the wave-shape in Fig. 3.

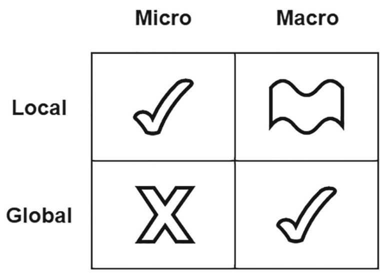


Fig. 3. Main design approach for organizing information into local and global percepts for micro (unit) or macro (manager) entities.

The ease of use of the percepts for an agent programmer should also be taken into consideration, i.e., by grouping related pieces of information together. The design guideline here is that one should *only group sets of parameters that naturally belong together*. Moreover, to avoid having to deal with different kinds of percepts for each type of unit, a design guideline is that *the percepts should be as generic as possible in order to facilitate re-use* between different agents. This guideline is aimed at reducing the number of different concepts introduced in our percept ontology, and thus aims for efficiency of design. An example of this is the `status` percept for each unit, as its structure (i.e., the set of parameters) is the same for each unit, even though not all information might be relevant for each unit (not all units use energy for example, but a unit's energy level is always provided in the percept). This also allows for specifying generic code for handling the `status` percept for all agents only once in the program, instead of having to specify this specifically (and nearly identically) for each unit type; special cases for certain types of units can then be programmed only where necessary.

Performance. One of the main challenges is how to deliver all percepts while guaranteeing sufficient performance levels. It is important to manage the percept load of individual agents, as creating the information needed for percepts (i.e., in the connector) and relaying that information to one or multiple agents who then have to make this information available for use in reasoning (i.e., by representing them in a Prolog base) is the most resource intensive task in a connector. In contrast to actions, of which usually at most one is selected per decision cycle, there are usually many percepts (all containing various amounts of information) sent to each agent per decision cycle. We therefore introduce a number of optimization guidelines which aim to either reduce the total number of percepts an agent will have (to store) or the amount of updates to this set of percepts that an agent will have to process.

Complex environments have a lot of static information to which all individual agents may need to have access, like what a certain unit costs to produce or what kinds of units a certain building can produce in StarCraft. Because such environments also introduce many units (and thus many agents), the initialization costs for such information for each of these agents can have a rather big impact on a connector's performance. To avoid this issue as much as possible, we introduce another design guideline to *only create percepts for information that changes in a single match or between matches*. Static information is better suited to be encoded in the agent system itself instead of being sent through percepts, as this will significantly reduce the performance when initializing an agent (which as aforementioned can happen many times during a game as large numbers of units come and go almost constantly). To this end, information that is fixed by the game itself can be coded as a separate part of the ontology that can and needs to be loaded only once at the start of the game. Agents will still need to be informed about changes between matches, e.g., map-specific information should not be included in the 'fixed part' of the ontology. Another guideline to keep the number of percepts low is to ensure that *no data is sent through percepts that can either be calculated based on other data* (e.g., the number of friendly units by counting the number of percepts about their status), *or retrieved from other agents* (e.g., the position of a friendly unit). Relaying information (like friendly unit positions) through messaging between the agents in a MAS is usually much more efficient, as an agent programmer can then selectively choose at which times and to which units to send specific pieces of information, as opposed to percepts always being sent to certain units even when they do not require them (at that time).

In order to improve the performance of the percepts that we do have to send, the Environment Interface Standard (EIS) [1], that we have used as a foundation for implementing our connector, differentiates between three types of percepts[1]:

- **Send once:** this type of percept is only sent once. Such percepts are generally used to send data about the (specific) match when an agent is created, such as information about the map on which the match is played.
- **Send on change:** a percept of this type will only be sent if the percept changes. Such percepts are generally used to update known information, such as a unit's health or the number of available resources.
- **Send always:** a percept of this type will be perceived every time the corresponding agent asks for percepts. Such percepts are generally used to indicate temporary information, such as seeing an enemy unit (which can die, after which the corresponding percept is no longer generated).

Send once percepts will be most performant, whilst send always percepts will be least performant. However, as indicated, some information cannot be represented in a 'more performant' type. It is thus important for to carefully consider which percept category certain (groups of) information would best fit in order to optimize the performance.

For StarCraft, combining the (finite set of) information that is available through the BWAPI interface with the guidelines as posed in this section lead to a set of about 25 percepts[2].

We have designed and optimized our algorithms to compute the difference between information states in order to generate new percepts as fast as possible. Most percepts are only generated if some change occurred. Our connector has been carefully designed so as to optimize the generation of percepts by first and only once generating the global percepts (i.e., that are not specific to units), such as the list of (visible) friendly and enemy units, followed by the generation of the percepts specific to each entity. This structure also ensures that agents receive their percepts immediately when they ask for them, i.e., they are not generated when requested (which would slow down the agent significantly) but only when information actually changes.

4.3 Asynchronous Actions

The actions available for a certain entity define the range of behaviour that is possible for a corresponding agent implementation. The basic design guideline here is that as a rule, *any action that a unit can do* (i.e., that is available in the environment) should be available to its corresponding entity (and thus agent).

[1] There are actually four percept types, but we do not consider *on-change-with-negation* as this type will be removed in future versions of EIS due to compatibility issues with knowledge representation languages other than Prolog.

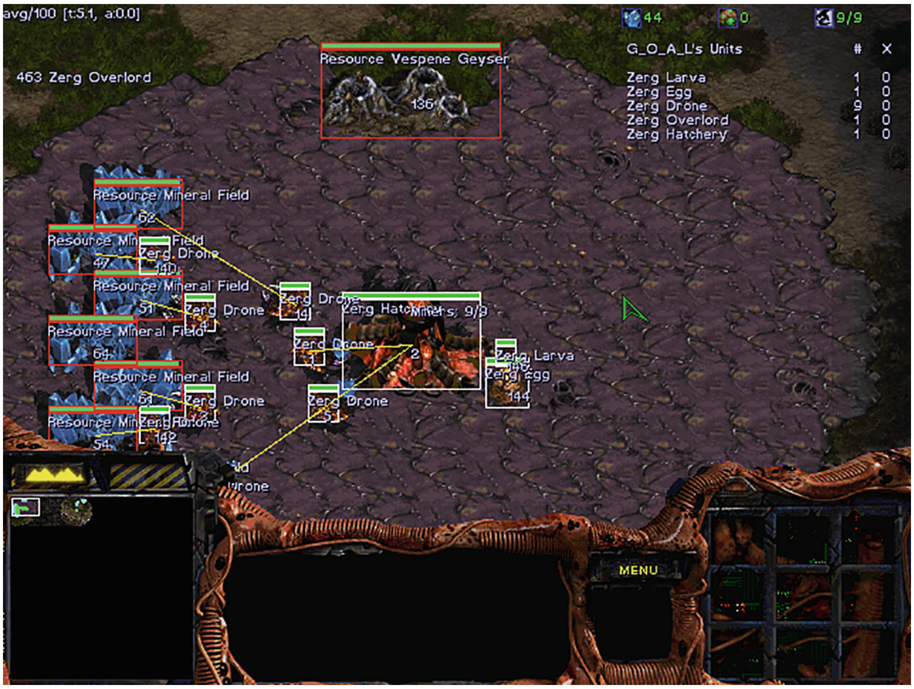
[2] For the full set of percepts and actions that are available, we refer to the StarCraft Connector Manual at https://github.com/eishub/Starcraft/blob/master/doc/Resources/StarCraftEnvironmentManual.pdf.

A unit in StarCraft can roughly choose from about 15 types of actions at any given time. Certain actions are only available to specific types of units (e.g., loading a unit into a loadable building). Some abstractions were used in order to better facilitate the usability of this set of actions for agent programmers. For example, instead of using pixel coordinates, StarCraft allows tile coordinates to be used, i.e., corresponding to a certain block of 32 by 32 pixels (buildings in StarCraft always have a size that is a multiple of 32 pixels in any dimension). This abstraction of pixels to tiles is also used in coordinates in percepts, thus not only ensuring easy compatibility with the actions but also allowing for percepts containing coordinates to be updated significantly fewer times when a unit is moving for example. We also note that BWAPI does not explicitly support grouping units (i.e., as a human player would do), and thus each unit needs to choose its own course of action. However, creating group behaviour in a multi-agent system is facilitated through inherent mechanisms such as messaging between agents. Manager agents thus do not need specific actions from a connector, as they can rely solely on the facilities in the agent platform.

However, as a MAS platform uses and runs agents in its own (set of) thread(s) that need to be connected to the environment, synchronisation issues arise that in particular for StarCraft pose a challenge, as StarCraft runs at a specific rate, updating the game logic at fixed millisecond intervals in so called '*match frames*'. In existing (C++/Java) BWAPI bots, the match frame function is used as the starting point (or even single function) for all decision making. In principle, this conflicts with a multi-agent approach in which all cognitive agents run in their own separate (autonomous) thread(s). As a solution, we use several synchronisation mechanisms. First, and most importantly, for each entity all requested actions are recorded (queued). On each match frame call, all queued actions (for all entities) are executed, i.e., 'forwarded' to the corresponding unit in StarCraft itself. Agents have to carefully rely on feedback from the environment (i.e., through percepts) to detect the effect of their actions, or when an action has failed (e.g., because some other action by another agent just used up some resources). A basic understanding of the synchronisation issues is thus needed when developing agents for highly dynamic environments such as StarCraft.

Debugging and Testing. For complex environments such as real-time strategy games and StarCraft in particular, it is also essential to provide a developer with environment-specific visualization tooling that provides easy access to information that will allow the developer to understand what is going on in this environment. Which (types of) tooling can be provided is specific to an environment and the access provided by the basic API made available by the environment.

In our case study, we have found that visualization tooling is most useful for providing insight into basic capabilities such as navigation, the status of units, and the progress of long-term actions such as a buildings producing a unit. For example, even though agents do not exercise low-level navigation control, agents do control the setting of target locations where units will move to. We therefore provide a developer with the option to enable visual cues about where a unit is moving to in order to be able to debug the agent code that sets these target locations. Another example of what our connector supports is visualizing when a unit is being produced by a building, removing the need to click on each building to see what it is producing (and how far along this production is) when trying to debug the production logic in a specific building agent. Visualizations like this can be implemented in StarCraft by using its debug drawing features that support drawing lines or writing text in the game window. Using these basic features, our connector allows for specific visualizations to be created by agents themselves (i.e., through calling specific actions), also facilitating drawing custom texts above in-game units. Examples of such 'debug visualizations' in StarCraft are shown in Fig. 4.



G_O_A_L's Units

	#	X
Zerg Larva	1	0
Zerg Egg	1	0
Zerg Drone	9	0
Zerg Overlord	1	0
Zerg Hatchery	1	0

Fig. 4. A screenshot of StarCraft with a bot performing many debug draw actions.

More generally, to be able to debug and test multi-agent systems effectively and efficiently in an environment such as StarCraft where hundreds of agents are running simultaneously, requires a developer to have access to cheats that disclose or even modify gaming information that is not normally available to a player. StarCraft specifically offers useful development functionalities (through BWAPI calls) like removing the fog of war (i.e., making the whole map visible to the player), quickly gaining resources, or to making units invincible. We have integrated these functionalities in a separate development tool (that includes a button for gaining resources for example) and through initialization properties of the connector (e.g., making units invincible right from the start of a match) in order to make them easily accessible.

4.4 Evaluation

As high performance is critical for any cognitive approach that uses many agents to deal with the challenges of AI for RTS, it is important to verify the (CPU) performance of a connector. In addition, one should evaluate the requirement that a connector does not restrict the strategy space in any essential way by for example examining the success (i.e., in tasks in the environment) of a set of cognitive MAS implementations that make use of the connector. We do so by discussing the lessons we learned from the use of our connector by over 500 students in two years.

Performance. Complex real-time tasks, such as effectively attacking enemy units in StarCraft, potentially require a new decision to be made in each match frame (based on the new information such a frame generates). As our approach is based on an unit-agent mapping, there are at least as many agents as units in the game. To be performant, we need to show that all agents have the opportunity to receive new percepts and make a decision (i.e., perform an action) each match frame. AI tournaments run StarCraft at speeds of at least 50 match frames per second, which implies that in our case every agent should receive new information and be able to perform a new action at least 50 times per second as well, i.e., averaging[3] at most 20 ms for performing all cycles of the agents in a MAS. We assume here that no single agent should perform less than 50 decision cycles per second, even though many agents will not need that many decision cycles (e.g., most buildings would not as the decision making for production is not as time critical as for combat for instance). We aim to demonstrate that the minimum load required in the execution of the StarCraft connector leaves sufficient CPU time for adding the key decision logic in an agent program. We do so for our StarCraft connector by evaluating a simple multi-agent system that keeps producing simple units ('Zerglings') that continuously move to a random location on the map. In addition, all of these units are subscribed to all percepts (i.e., have to process them every decision cycle). A cheat was also enabled to ensure that these units cannot die. In this way, the maximum amount of units that a player can have (which is close to 400) can be reached without being influenced by the enemy in the game. Even though a player is very unlikely to reach this number of units in a game in practice, or to have all units subscribed to all percepts, we aim for our connector to provide sufficient CPU time for strategic reasoning even in this worst-case scenario.

[3] Most tournaments allow bots to take more time for a limited amount of frames during a single match, but we disregard that here.

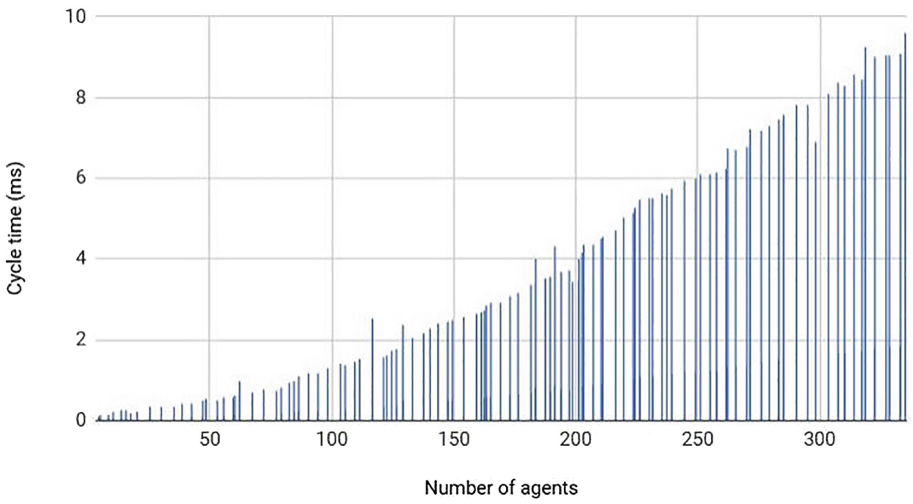


Fig. 5. The average speed of a decision cycle for all agents in the system under a growing number of agents.

The results of this evaluation for a minimum baseline are shown in Fig. 5. The evaluation was performed on a system with an Intel i7-6500U CPU and 8 GB RAM, with the StarCraft game speed set to the default tournament speed of 50 FPS. As each agent runs in its own thread(s), the average time any agent's cycle takes will increase when the number of agents increases due to limited system resources (e.g., the number of available CPU cores). However, even in this worst-case situation with up to 400 agents all processing all information available in the environment, the average cycle time per agent grows to about 10 ms at most. This thus leaves 10 ms for any additional reasoning to be implemented in the MAS in this extreme scenario. In practice, there will be fewer agents that are all subscribed to percepts more selectively. Therefore, in general, we see that around 18 ms (out of the possible 20 ms enforced by the tournaments themselves) will be available to a MAS that uses our connector.

We note that we have designed this baseline MAS such that all of the agents continuously execute decision cycles, whilst in practice, a decision is not required by each agent in every frame. This fact provides further support for our claim that sufficient processing power remains for implementing decision logic, as agents in a MAS with a more diverse set of agents should refrain from executing decision cycles (i.e., 'sleep') from time to time, thus freeing up CPU time for where it is needed most.

Success. As we cannot directly establish whether the full strategy space is made available by a connector, we aim to indirectly determine this by how well a cognitive MAS is able to perform relative to an environment measure that we would like to optimize. For a game like StarCraft, being successful at the game by winning (against other AI implementations) can provide such a measure.

Over the course of two years, groups of students created a varied range of full-fledged StarCraft AI implementations using (different versions of) our connector. After at most 8 weeks of work, nearly all of their implementations are able to defeat the game's built-in AI consistently. Some of the groups joined the Student StarCraft AI Tournament (SSCAIT) [20] with their implementations, successfully competing with the over 100 other active bots (which are mostly written in C++ or Java, frequently based on other well-established implementations, and have often been around for many years or developed by companies like Facebook). One of the students' StarCraft AI implementations that makes use of our connector is currently ranked at around the 50th place with a win-rate of roughly 60%. Altogether, this suggest that we have made the strategy space associated with StarCraft sufficiently available.

During the development and initial uses of the connector, we also gained valuable insights into the benefits and challenges of using current cognitive technologies for engineering complex distributed systems. One particularly challenging development issue that developers face when environments become more complex and the number of agents increases, is that every run of the system will produce different results. For this reason, it is very hard for a developer to test a specific scenario that s/he has in mind without additional tooling to provide a developer with control over the type of scenario that will evolve in the game. This makes testing very difficult and it thus is of the utmost importance to do whatever possible to provide a developer with tooling and capabilities to handle this. Testing against StarCraft's built-in AI, for example, will give different results on each run. More importantly, it can take quite a while before a scenario of interest occurs (if it does at all). In order to test specific (defined) scenarios, agent programmers should be allowed to *save the state of the game* at any given point, and then load that specific game again at a later stage, which is supported by StarCraft itself. Although our connector has been designed to support such state saving, in practice this will only provide support to some extent, and agent platforms should provide some way of storing and restoring the state of all agents at the same time.

4.5 Impact on Cognitive Technology

Even though the StarCraft connector has been optimized as far as possible when it comes to percept delivery, we found that there still are optimizations that can and should only be provided by the cognitive technology that is used, as we can only do so much; if the MAS platform itself is inefficient, it will not be possible to create an effective MAS approach for StarCraft with its strict real-time response requirements. One issue is for example that cognitive agents typically try to run as many decision cycles as possible. Considering the large number of agents that are typically employed in StarCraft, however, this is not ideal. In order to free up cycle time for e.g. agents that that have received new information to reason about. Therefore, we believe that functionalities that reduce the total load on the CPU, such as a '*sleep mode*' in which an agent that does not receive new percepts from the connector or new messages from other agents will not execute any reasoning, should be provided by agent platforms.

However, problems do arise in this mode when for example an agent is supposed to do something (e.g., move around) after it has not received new information for some time. Therefore, a *timing mechanism* should be introduced as well, facilitating the automatic generation of timer percepts upon a certain requested interval (thus waking up the agent after a set amount of time). A *sleep* action can be added as well, allowing a developer to manually sleep an agent for a certain amount of time, and thus free up performance for other agents if they do not need to do any reasoning for a while (even when new information comes in). An example of this is when a building agent starts producing a new unit, and is sure it will keep producing this unit (which takes a while). In addition, to allow developers to get more insight into the performance of their agents, specific logging messages can be added to agents that when enabled, after each decision cycle, show how many queries were performed and how many beliefs, goals, percepts and messages the agent has (received) in total. This can be useful for a developer to for example improve the ordering or nesting of rules in order to reduce the average amount of queries that are executed per cycle, or to keep tabs on the amount of messaging between agents (e.g., one agent might flood another agent with redundant messages due to some bug).

Another observation is that communication with large amounts of agents poses many challenges. In practice, with peer-to-peer based messaging, as is typically done in cognitive architectures, developers often use broadcasts to all agents in order to prevent having to use numerous bookkeepings of agents, which has an especially large performance impact in systems with many agents (such as those for StarCraft). We believe that this suggests that agent platforms should support a *publish-subscribe* messaging system to be effective, as this prevents agents that need to send messages to other agents from having to deal with continuously keeping track of which agents are relevant for its messages (i.e., interested in the information and still alive). Publish-subscribe messaging facilitates sending messages to a '*channel*'. Agents can subscribe to (and unsubscribe from) such channels, thus receiving messages sent to a certain channel only if they have explicitly indicated they want to do so. This allows for messaging based on content instead of specific targets. This is especially convenient for 'manager agents' to communicate with other (groups of) agents, as such an agent could for instance relay all required information about enemy units in a specific region to a certain channel, to which agents that need that information can then subscribe.

We believe that the application of cognitive agent technologies to complex environments such as StarCraft will yield more ideas for further development.

5 Conclusions and Future Work

We have presented a design approach for creating connectors for cognitive agent technology to (complex) environments, illustrated by a case study of such a connector that provides full access to StarCraft. A major challenge that was addressed during the development of this connector was to ensure corresponding cognitive agent systems can be programmed at a high level of abstraction whilst simultaneously allowing sufficient variety in strategies to be implemented by such systems. Based on this challenge, design guidelines for determining the set of available percepts and actions in agent-environment connectors were determined. The viability of our approach is demonstrated by multiple large-scale practical uses of the StarCraft connector, resulting in a varied set of competitive AIs. Based on the development of the connector and this initial use, we gained valuable insights such as the benefits of using publish-subscribe based messaging and the challenges of debugging large sets of agents.

Ensuring a sufficient level of performance of the connector was a significant challenge that had to be addressed in particular in order to demonstrate that a unit-agent mapping (MAS) approach is viable. In our evaluations, we determined the baseline performance of the connector in a worst-case scenario, which shows that on average there remains sufficient CPU time for strategic reasoning in a cognitive MAS. Even though the performance of such a MAS depends largely on the agent technology used itself, we believe that our connector can be effectively used in practice. Although our case study is focused on the 'Brood War' version of StarCraft, the new 'raw API' of StarCraft 2 is reported to be similar to BWPAI by Vinyals et al. [18], and tour work should therefore be relatively straightforwardly applicable and/or portable to StarCraft 2 (and possibly other RTS games) in future work.

Finally, through the development and use of our connector for StarCraft, a number of challenges to cognitive agent technologies were identified. One of those challenges is the fact that debugging (cf. Koeman et al. [12]) becomes increasingly difficult with increasing numbers of agents. As debugging concurrent programs is a hard problem in general, more work is required in this area; it could for example be useful to visualize the interaction between agents or the CPU time required by each agent. In addition, in order to better support automated testing, (cf. Koeman et al. [11]), it may be beneficial to develop a mechanism that automatically saves the state of a MAS when a save game is created in StarCraft. This can be used to immediately initialize a MAS to the desired state when executing a test with a specific save game (i.e., a scenario). Another observation is that communication with large amounts of agents poses many challenges, requiring more investigation in future work, for example into messaging architectures based on a publish-subscribe pattern. Finally, the performance of a MAS itself (i.e., all processing that takes place outside of a connector) is of critical importance in highly dynamic environments such as StarCraft. Functionalities that can reduce the CPU load of a MAS are thus important to explore as well.

References

1. Behrens, T.M., Hindriks, K.V., Dix, J.: Towards an environment interface standard for agent platforms. Ann. Math. Artif. Intell. **61**(4), 261–295 (2011)
2. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Tools and Applications. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-89299-3

3. Dignum, F.: Agents for games and simulations. Auton. Agents Multi-Agent Syst. **24**(2), 217–220 (2012)
4. Dignum, F., Westra, J., van Doesburg, W.A., Harbers, M.: Games and agents: designing intelligent gameplay. Int. J. Comput. Games Technol. **2009**, Article ID 837095 (2009)
5. Heinermann, A.: Brood War API (2008). https://github.com/bwapi/bwapi. Accessed 12 May 2018
6. Hindriks, K.V.: The shaping of the agent-oriented mindset. In: Dalpiaz, F., Dix, J., van Riemsdijk, M.B. (eds.) EMAS 2014. LNCS (LNAI), vol. 8758, pp. 1–14. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14484-9_1
7. Hindriks, K.V.: Unreal Goal Bots. In: Dignum, F. (ed.) AGS 2010. LNCS (LNAI), vol. 6525, pp. 1–18. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18181-8_1
8. Jennings, N.R., Sycara, K., Wooldridge, M.: A roadmap of agent research and development. Auton. Agents Multi-Agent Syst. **1**(1), 7–38 (1998)
9. Jensen, A.S., Kaysø-Rørdam, C., Villadsen, J.: Interfacing agents to real-time strategy games. In: SCAI, pp. 68–77 (2015)
10. Koeman, V.J., Griffioen, H.J., Plenge, D.C., Hindriks, K.V.: Designing a cognitive agent connector for complex environments: a case study with starcraft. In: Proceedings of the 6th International Workshop on Engineering Multi-Agent Systems. EMAS 2018, July 2018
11. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Automating failure detection in cognitive agent programs. In: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, AAMAS 2016, pp. 1237–1246. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2016)
12. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Designing a source-level debugger for cognitive agent programs. Auton. Agents Multi-Agent Syst. **31**(5), 941–970 (2017)
13. Lara-Cabrera, R., Cotta, C., Fernández-Leiva, A.: A review of computational intelligence in RTS games. In: 2013 IEEE Symposium on Foundations of Computational Intelligence (FOCI), pp. 114–121, April 2013
14. Logan, B.: A future for agent programming. In: Baldoni, M., Baresi, L., Dastani, M. (eds.) EMAS 2015. LNCS (LNAI), vol. 9318, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26184-3_1
15. Müller, J.P., Fischer, K.: Application impact of multi-agent systems and technologies: a survey. In: Shehory, O., Sturm, A. (eds.) Agent-Oriented Software Engineering, pp. 27–53. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54432-3_3
16. Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., Preuss, M.: A survey of real-time strategy game AI research and competition in StarCraft. IEEE Trans. Comput. Intell. AI Games **5**(4), 293–311 (2013)
17. Robertson, G., Watson, I.: A review of real-time strategy game AI. AI Mag. **35**(4), 75–104 (2014)
18. Vinyals, O., et al.: StarCraft II: a new challenge for reinforcement learning. arXiv preprint arXiv:1708.04782, August 2017
19. Weber, B.G., Mateas, M., Jhala, A.: Building human-level AI for real-time strategy games. In: AAAI Fall Symposium: Advances in Cognitive Systems, vol. 11 (2011)
20. Čertický, M., et al.: Student StarCraft AI Tournament (2011). https://sscaitournament.com. Accessed 12 May 2018