TODO: A Survey of StarCraft AI Techniques

FirstName LastName, *Member, IEEE,* Jim Raynor, *Fellow, RR,* and Sarah Kerrigan, *Life Fellow, ZS*

***Abstract*—TODO**

***Index Terms*—review, RTS, StarCraft, machine learning, planning, TODO ...**

I. Introduction

STARCRAFT AI competitions have caused many AI techniques to be applied to RTS AI. We will list and classify these approaches, explain their power and their downsides and conclude on what is left to achieve human-level RTS AI. TODO (test [**?**])

II. Challenges, why is it hard to do a good RTS AI?

Here we can use Buro's 2003 paper as a starting point. Much has changed since, so we should update, and put his predictions in perspective

III. AI Techniques review

A. Overview

B. Case study 1: EISBot

C. Case study 2: NOVA

D. Case study 3: BroodwarBotQ

IV. AI Architectures for RTS AI

Playing an RTS game involves dealing with all the problems described above. A few approaches, like CAT [**?**], Darmok [**?**] or ALisp [**?**] try to deal with the problem in a monolithic manner, by using a single AI technique. This resembles approaches to solve other games, such as Chess or Go, where a single game-tree search approach is enough to play the game at human level. However, none of those systems aims at achieving near human performance. In order to achieve human-level performance, RTS AI designers use a lot of domain knowledge in order to divide the task of playing the game into a collection of sub-problems, which can be dealt-with using individual AI techniques (as discussed in the previous section). Thus, an integration architecture is required to put together all of those techniques into a single coherent system, which is the focus of this section.

Figure 1 shows some representative examples of the integration architectures used by different bots in the AIIDE 2011 Starcraft AI competition [**?**]. Each box represents an individual module with a clearly defined task (only modules with a black background can send actions directly to Starcraft). Dashed arrows represent data flow, and solid arrows represent

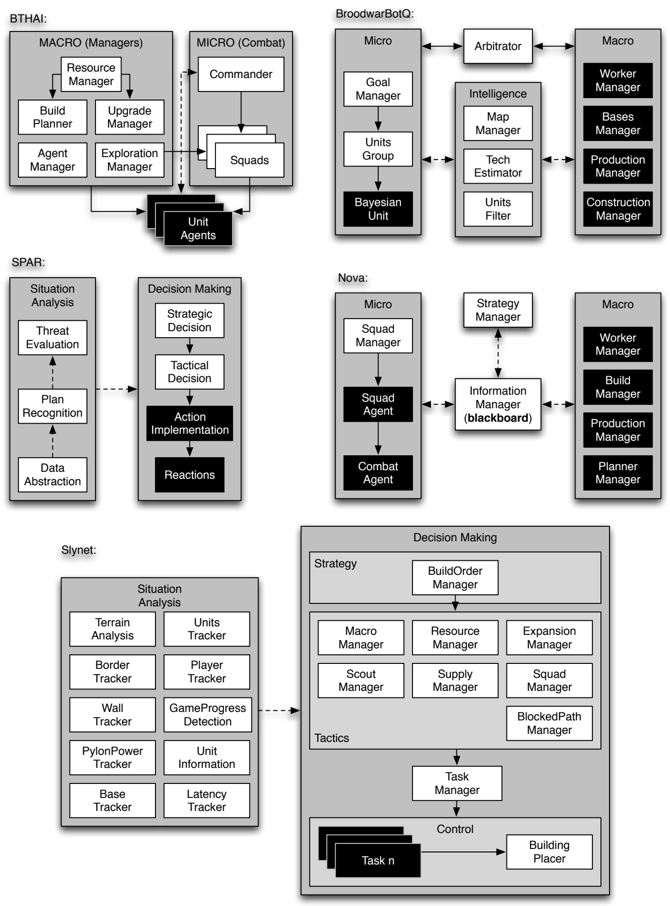

Fig. 1. General architecture of 5 Starcraft AI bots. (This figure is unreadable as of now, but we'll figure out a way to better present this info)

control (when a module can command another module to perform some task). For example, we can see how SPAR is divided in two sets of modules: *situation analysis* and *decision making*, the first with three modules dedicated to analyze the current situation of the game, and the later with 4 modules dedicated to exploit that information to decide what to do. We can see how the decision making aspect of SPAR is organized hierarchically, with the higher-level module (*strategic decision*) issuing commands to the next module (*tactical decision*), which sends commands to the next module (*action implementation*), and so on. Only the lower-level modules can send actions directly to Starcraft.

On the other hand, bots such as NOVA or BroodwarBotQ (BBQ) only use a hierarchical organization for *micro-management* (controlling the attack units), but use a decentralized organization for the rest of the bot. In Nova and BBQ, there is a collection of modules that control different aspects of the game (workers, production, construction, etc.). These

FirstName LastName is with the Department of Names GA, 30332 USA e-mail: (see http://www.michaelshell.org/contact.html).
J. Raynor and S. Kerrigane are with the Romeo&Juliet Inc.
Manuscript received April 19, 2499; revised January 11, 2500.

modules can all send actions directly to Starcraft. In Nova those modules coordinate only through writing data in a shared blackboard, and in BBQ they coordinate only when they have to use a shared resource (unit) by means of an arbitrator.

By analyzing the structure of these bots, we can see that there are two main tools that can be used when designing an integration architecture:

- *Abstraction*: complex tasks can be formulated at different levels of abstraction. For example, playing an RTS game can be seen as issuing individual low-level actions to each of the units in the game, or at a higher level, it can be seen as deploying a specific strategy (e.g. a "BBS strategy", or a "Reaver Drop" strategy). Some bots, reason at multiple levels of abstraction at the same time, making the task of playing Starcraft simpler. Assuming that each module in the architecture of a bot has a goal and determines some actions to achieve that goal, the actions determined by higher-level modules are considered as the goals of the lower level modules. In this way, each module can focus on reasoning at only one level of abstraction, thus, making the problem easier.
- *Divide-and-conquer*: playing a complex RTS, such as Starcraft, requires performing many conceptually different tasks, such as gathering resources, attacking, placing buildings, etc. Assuming each of these tasks can be performed relatively independently and without interference, we can have one module focusing on each of the tasks independently, thus making the problem easier.

If we imagine the different tasks to perform in a complex RTS game in a two-dimensional plane, where the vertical axis represents abstraction, and the horizontal axis represents the different aspects of the game (micro-management, resource gathering, etc.), abstraction can be seen as dividing the space with horizontal lines, whereas divide-and-conquer divides the space using vertical lines.

Different bots, use different combinations of these two tools. Looking back at Figure 1, we can see the following use of abstraction and divide-in-conquer in the bots:

- BTHAI: uses a two-tier abstraction hierarchy, where a collection of high-level modules command a collection of lower-level agents in charge of each of the units. At the high-level, BTHAI uses divide-and-conquer, having multiple high-level modules issuing commands to the lower-level units.
- BBQ: uses abstraction for micro-management, and divide-and-conquer for macro-management and intelligence gathering. To avoid conflicts between modules (since the individual tasks of each of the modules are not completely independent), BBQ uses an arbitrator.
- Nova: is similar in design as BBQ, and uses abstraction for micro-management, and divide-and-conquer for macro-management. The differences are that Nova does not have an arbitrator to resolve conflicts, but has a higher-level module (*strategy manager*), which posts information to the blackboard that the rest of modules follow (thus, making use of abstraction).
- SPAR: only uses abstraction. Its high-level module determines the strategy to use, and the tactical decision module divides it into a collection of *abstract actions*, that are executed by the lower-level modules.
- Skynet: makes extensive use of both abstraction and divide-and-conquer. Considering the decision making component of Skynet, we can see a high level module that issues commands to a series of tactics modules. The collection of tactic modules queue *tasks* (that are analogous to the abstract actions used in SPAR). Each different task has a specific low level module that knows how to execute it. Thus, Skynet uses a 3 layered abstraction hierarchy, and uses divide-and-conquer in all levels except the highest.

Additionally, except for BTHAI, all other agents use divide-and-conquer at a higher-level bot design and divide all the modules into two or three categories: *information gathering* and *decision*, or *information gathering*, *micro-management* and *macro-management*.

Notice that most bots using divide-and-conquer (except for BBQ), assume that each of the modules can act independently and that their actions can be executed without interference. BBQ is the exception, including an arbitrator that makes sure that modules do not send contradictory orders to the same unit. However, very little bots handle the problem of how to coordinate resource usage amongst modules, for instance BTHAI uses a first-come-first-serve policy for spending resources, the first module that requests resources is the one that gets them. The exceptions are Nova and Skynet implement some rudimentary prioritization based on the high level strategy.

One interesting aspect of the five bots described above is that, while all of them are reactive at the lower level (unit control), most if not all of them, are scripted at the highest level of abstraction. BTHAI reads build and squad formations from a predefined script, Nova's *Strategy Manager* is a predefined finite-state machine, BBQ's construction manager reads the build order from a predefined script, and Skynet's *BuildOrder Manager* is basically a predefined script. Such scripts describe the strategy that the bots will use, however, such strategy is always fixed. One could see this pre-scripting as if each bot defined a "high-level programming language" to describe Starcraft strategies, and the bots themselves are just interpreters of such strategy. Compared to current approaches for Chess or Go, this scripting seems a bit rigid and inflexible, but responds to the much higher complexity of the Starcraft game.

In conclusion, we can see that there are two basic tools that can be used in an integration architecture: abstraction and divide-and-conquer, which are widely used by the existing Starcraft bots. However, several open questions remain:

- What is the best task decomposition for RTS games?
- Resource coordination in a divide-and-conquer approaches.
- Reactive high-level strategies that do not depend on scripting.
- ...

V. Discussion, what is left to do / open challenges

VI. Conclusion

Acknowledgment

Jim Raynor Jim Raynor was a Confederate marshal on Mar Sara at the time of the first zerg incursions on that world. He is now with Raynor's Raiders Inc.