# Problem decomposition and learning integration

Decision-making is just one of many competencies required by a bot to be able to play games like StarCraft. In {chapter link} we presented various other skills vital for successful bot and showed that those skills might have certain depth as it is in the case of decision making. To create a capable bot for games like StarCraft one must use many techniques presented in {chapter link}. Inverse Reinforcement Learning is just one of the techniques to obtain one particular skill, and it alone is not enough to realize bot. Therefore we developed an opinionated framework based on ideas of distributed multi-agent systems described in {chaper link} to be able to integrate other techniques to our bot and due to the necessity to decompose the problem of playing StarCraft to more manageable subproblems. In the following chapter, we present our framework, the high-level architecture of our bot and integration of learning. The reasoning behind using MAS can be found at the end of the previous {chapter}. All code related to our work can be found in public repository {link}.

## Framework

First of all, it is important to explain why we bothered developing something such as framework in the first place when many others bots such as [1] and [2] use MAS ideas to some extent, and even some previous works aim at developing framework of some kind [3]. Simply it was not enough for our use case as those works are even more opinionated than our intended framework. In our opinion, it would put many restrictions on the future development as they do not meet our requirements:

* We do not want to enforce some static form of structure as relations between subproblems are dynamic and vary trough the time and situation. We want for parts of the system to make proposals of the goals that they think are right for the collective and augment the environment for others. Others can then decide their level of involvement and behavior for the situation. We want to have our components as simple as possible by reducing communication and focusing more on properties of the environment. This kind of contract introduce a loose way of coupling for individual components, enables greater abstraction with better decomposition and make problems simple.
* We want our component to be able to change its behavior based on circumstances. As we are dealing with complex problems, we want to make things easier by reasoning what our search space looks like first and then plan on this space.
* We want our framework to be modular and easily extensible with the possibility to integrate many techniques.
* We want for our framework to be portable; we may not want to use it just for StarCraft.

In our framework, we use terms **Beliefs, Desire, and Intention** architecture from **BDI** architecture [4] to model agents. It is one way how to model rational agents. From now on, to beliefs we refer as the information set available to the agent, desires are objectives which agent may want to achieve, and intentions are objectives agent has committed to accomplish. Intention also contains a **plan** to reach the goal. To illustrate the relation between those three terms we give following situation from StarCraft where the player has many options when choosing units to build (desires). Based on his beliefs about his and enemy army composition (beliefs) he starts to create a particular unit mix to adapt his army (intentions).

### Architecture overview and basic properties

Based on our requirements we designed our framework as distributed multi-agent system. High-level architecture overview is on figure {link} where building blocks of our system are shown. Main components of the system are agents and mediators for desires and beliefs. The system is accessible trough façade.

{figure – architecture - framework}

The central part of our framework is an agent. Its implementation is in alignment with agent’s principles outlined in {link}. It has an own memory to represent beliefs where it stores data from sensors of the environment, additional facts it may deduce and accessible beliefs shared by other agents trough mediator. An important part of the memory is a belief about its current plan or plans of others. The goal for agent design was to be as simple as possible to ease complexity and enable better decomposition. We made communication as part of the environment – trough desire and belief mediator.

Each agent treats same way own and desires periodically consumed from the mediator. The agent commits to desire based on current settings and state of beliefs; it is only reacting to the situation of the environment. Therefore agents are very loosely coupled and simple. The way how agents make proposals to the system is based on **Blackboard** architecture [4]. In this architecture, experts share expertise by stating what may be good for collective to let others do that. Bot Nova [2] also uses this kind of architecture. Using this architecture allows topology of the agents to be dynamic as agents decide which desires to follow and only than hierarchy is introduced.

### Planning, decision making, and techniques integration

We design our planning as another way how to do problem decomposition. Our approach is similar to Behaviour Trees or Hierarchical Finite State Automat with the difference that we simplify the structure and add the ability to create structure dynamically. The architecture of our planning algorithm is on figure {link}. Each agent contains component referred to as “**HeapOfPlanningTrees**” which has set of trees to represent search space for desires and their realizations. Each planning tree is composed of modules where each instance is desire or intention. When a module has a form of intention, it defines an internal action for an agent to take – to reason about something, execute some action in the environment, share desire with the system or extend current tree by adding additional nodes. On top of that, each agent has its lookup library to choose appropriate intention for each desire it has. Selection of desire is currently based only on key-value lookup where current desire and its parent is the key. Replacing key-value approach by modular one in the future will grant us with the possibility to integrate other techniques to improve search for right behavior in given situation.

Every module is declarable by the user, so there are endless possibilities how to implement new techniques or to further decompose the problem. For deliberation, we use submodules named as “**Commitment Decider**” which implementation is only up to the user and is part of each module. Those submodules serve as decision-making points, and those are the places where we use **Inverse Reinforcement Learning or other techniques** such as hard-coded rules to decide when to commit to desire or to stop follow intention. Every decision can be based on beliefs and structures of trees.

{figure – planning architecture}

Various visitors periodically visit every tree – two to decide commitment and third to execute executable content of terminal modules. The way how they do that is shown on figure {link} which represent an abstract overview of agent routine. User-defined configuration initializes each agent by declaring desires agent may want to pursue, beliefs it works with and lookup library to build up trees. The important part of the process is to keep most current beliefs and desires as they are a vital part of agent’s decisions.

{figure – routine process}

### Implementation of framework

We implemented our framework as domain independent where agents run in parallel synchronizing only trough mediators. To achieve good performance and robustness communication with mediators is asynchronous. Each mediator keeps a queue of sharing requests which are incorporated to working register. Mediator also maintains a read-only version of register available for agents. This register is periodically updated by working one in a way that it can be accessed concurrently by agents. The downside of this approach is the fact that data contained may not be most recent. However, this is the obvious downside of most distributed systems.

As the language of implementation, we choose **Java 8**. Java is portable and strongly typed Object-Oriented programming language. On top of that, it supports generic programming and has some functional features. Together with good dependency management and a lot of available libraries for our use case, it makes an excellent choice. Only one downside for usage of Java is a lack of multiple inheritance support.

## Integrating Learning

To develop bot and integrate Inverse Reinforcement Learning to it, we add to our project three different packages. Executable packages are Replay Parser and Bot. Both share package Abstract Bot. We present those packages together with our implementation specifics and way how we Integrate Inverse Reinforcement Learning in the following text.

### Abstract Bot

As our framework is domain independent, we need to introduced another layer to our bot to be able to play StarCraft. Abstract Bot contains wrappers for StarCraft objects which we are accessing trough **BWMirror** API {link} written in Java. This library maps on **BWAPI** {link} connected to the game. Due to properties of our framework, we had to write wrappers for most of the objects we use from BWMirror as it does not support concurrent access and does not allow to work with objects outside of the main routine. We accompanied wrappers by the cache to store wrapped instances of game-related objects. There are other performance benefits of it. As agents access game independently – we can serve them cached objects already accessed by different actors.

Another important part of Abstract Bot package are declarations for two remaining packages such as types of agents, their desires, feature container headers to describe IRL states, and beliefs. We use common declarations to make sure that data used in both packages check.

### Replay Parser

The Replay Parser package contains two executable programs. One is for game observation, and other is to learn decision-making trough Inverse Reinforcement Learning using those observations. The part to observe the replays consists of observers representing agents we would like to learn to make decisions based on observation of players. Each observer has predefined set of desires to track. It tracks values of features defined in “**FeatureContainerHeader**” to describe the state, and decisions made in the form of trajectory. After each replay, trajectories are saved.

To learn decision making, we use the second program. For each agent and type of desire, it loads saved trajectories. To be able to use IRL we have to reduce the number of states from trajectories. As a baseline for state compression, we use K-Means clustering with Euclidian distance, Z-Score normalization and the user-defined number of clusters. For clustering, we employ JSAT library {link} which provides an implementation of Minibatch K-Means [5]. Our approach is currently very biased and far from optimal but computationally bearable in our setting. Using states and trajectories, we then can create MDP with additional dummy state. We use this state as a destination for transitions which did not take place in replays. During the run of IRL algorithm, we keep reward for this state as small as possible to learn a policy which will not use unknown transitions. For IRL we extended algorithms and MDP implementations of BURLAP library {link} to match our use-case. The learned policy is then saved as decision module which is then loaded by bot.

### Bot

The figure {link} represents high-level architecture overview of our bot and shows the integration of components. It contains package Bot with the concrete implementation of agents –units, abstract agents and even agents for places. We present additional implementation details on our agents in next chapter.

{figure – bot architecture}

An important part of this package is “**BotFacade**” implementing method for various events called by the game. This way bot can communicate with the game by issuing commands or reading data. To do that we introduced “**GameCommandExecutor**” which manages requests of individual agents on game trough queue. It has a time window for handling requests on game call. It also keeps execution time of the request type to plan ahead as it makes sure that time will not be exceeded.

# Bibliografie

|  |  |
| --- | --- |
| [1] | BEN G. WEBER., . *Integrating learning in a multi-scale agent* [online]. 2012. Santa Cruz: University of California, 2012 [cit. 2017-01-11]. ISBN 14-776-1473-7. Dostupné z: http://alumni.soe.ucsc.edu/~bweber/bweber-dissertation.pdf |
| [2] | PÉREZ, Alberto Uriarte. *Multi-Reactive Planning for Real-Time Strategy Games*. Barcelona, 2011. Universitat Autònoma de Barcelona. |
| [3] | FIEDLER, David. *Použití metod multiagantních systémů pro implementaci umělé inteligence v real-time strategiích*. Plzeň, 2016. Západočeská univerzita v Plzni. |
| [4] | EDITED BY GERHARD WEISS., . *Multiagent systems: a modern approach to distributed artificial intelligence*. Reprint. Cambridge, Massachusetts: The MIT Press, 2001. ISBN 978-026-2731-317. |
| [5] | SCULLEY, D. Web-scale k-means clustering. *Proceedings of the 19th international conference on World wide web - WWW '10* [online]. New York, New York, USA: ACM Press, 2010, , 1177- [cit. 2017-05-07]. DOI: 10.1145/1772690.1772862. ISBN 9781605587998. Dostupné z: http://portal.acm.org/citation.cfm?doid=1772690.1772862 |