SARL Agents in City Controller

Report by
Karthi Narendrababu Geetha(s3835901)
Prabhat Kumar Singh(s3833321)
Varsha Chandrasekhar Bendre(s3831858)

Table of Contents

General Strategy for the Game

The main objective of “Agents in City Controller” is to complete jobs and earn money by buying, assembling and delivering items. A significant amount of the strategy is adapted from the team BusyBeaver [1]. Figure 1 demonstrates the strategy of the project. The agents are grouped into Suppliers, Builders and Deliverers. The Leader agent spawns all three agents. The Suppliers buys the base items from the shops and give them to the Builders to assemble. The Builders assemble the advanced items from the base items with the help of builder tools. The Deliverers collect the items from the builders and deliver the items to the storage.

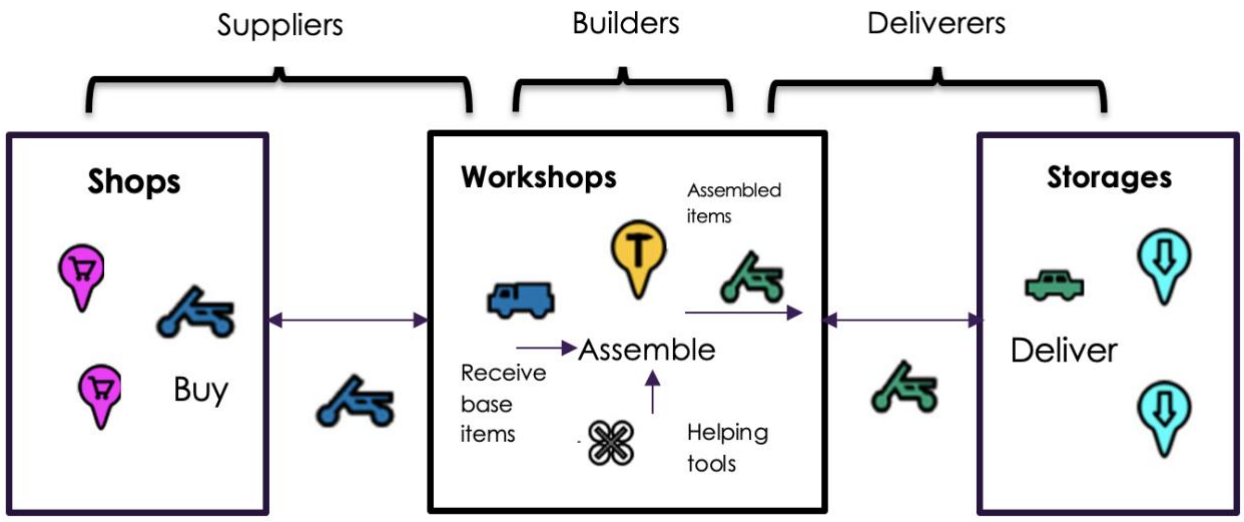


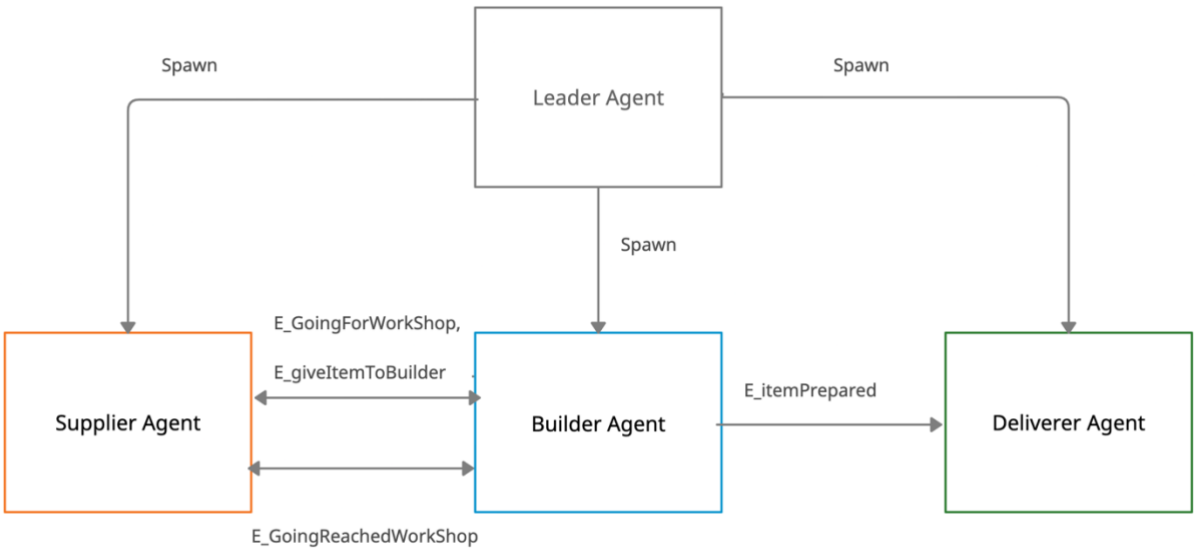
Figure 1: The Strategy of Agents in City Controller

The distribution table for the number of roles in each agent is depicted in Table 1. All the agents interact with each other at the workshop and most of the actions are coordinated by the Supplier agents. The agents move to the charging station when the battery is low, and then continue with their respective jobs. The Resource nodes were not considered, as this project was not a part of any competition.

	Suppliers	Builders	Deliverers
Drones	1	1	2
Motorcycles	3	1	4
Cars	4	1	5
Trucks	3	2	4

Table 1: The distribution table for the roles

Architecture



Agents

Leader Agent

As shown in the above diagram, the leader agent is responsible for spawning other agents. Do to so, we are using the three configuration file: *buildermassimconfig.json* (code snippet shown below), and *suppliermassimconfig.json*, *deliverermassimconfig.json* .

```
{
  "scenario": "city2018",
  "host": "localhost",
  "port": 12300,
  "scheduling": true,
  "timeout": 40000,
  "times": false,
  "notifications": false,
  "queued": false,
  "entities": [
    {"name": "entityA2", "username": "agentA2", "password": "1", "iilang": false, "xml": true},
    {"name": "entityA6", "username": "agentA6", "password": "1", "iilang": false, "xml": true},
    {"name": "entityA14", "username": "agentA14", "password": "1", "iilang": false, "xml": true},
    {"name": "entityA25", "username": "agentA25", "password": "1", "iilang": false, "xml": true},
    {"name": "entityA29", "username": "agentA29", "password": "1", "iilang": false, "xml": true}
  ]
}
```

Supplier Agent

The primary goal of the supplier agent is to collect items required for completing a given job. For that, the entities assigned to this agent performs various actions like buying items from the shop, delivering items to the builder agent etc, by emitting the events E_GoingForWorkShop and E_giveItemToBuilder.

Builder Agent

The builder agent is used for assembling the items provided by the supplier agent. Once all the items required for a given job are assembled, it emits event E_ItemPrepared to notify the deliverer agent to collect these items from it.

Deliverer Agent

The deliverer agents are used for delivering the items collected from builder agents to the storage location mentioned inside the job. By doing so, a given job is finally completed.

Events

E_GoingForWorkShop

```
event E_GoingForWorkShop {

    var workShopName : String
    var entityName : String

    new(workShopName : String, entityName : String) {
        this.workShopName = workShopName
        this.entityName = entityName
    }
}
```

i. entityName: The supplier entity which is going towards the workshop
ii. workshop: Workshop name where the entity is headed towards.

This event is used by the Supplier agent to notify the builder that it is moving towards the workshop so that the builder agent sends its entities there as well to prevent delay in sending and receiving items.

• • •

E_ReachedWorkShop

```
event E_ReachedWorkShop {

    var workShopName : String
    var agentName : String
    var availableLoad : int
    var entityAgentType : String

    new(workShopName : String, agentName : String, availableLoad : int, entityAgentType : String)
        this.workShopName = workShopName
        this.agentName = agentName
        this.availableLoad = availableLoad
        this.entityAgentType = entityAgentType
    }
}
```

i. agentName: name of the builder entity which has reached the workshop
ii. workShopName: name of the workshop where the entity has reached
iii. availableLoad: the difference between maximum load and current load of an entity
iv. entityAgentType: its value can be builder, supplier or deliverer based on the agent's name to whom the entity belongs to

This event is emitted by the builder agent to notify the supplier agent that it has reached the workshop and is ready to receive items.

E_giveItemBuilder

```
event E_giveItemToBuilder {
    var builderAgentName : String

    new(builderAgentName : String) {
        this.builderAgentName = builderAgentName
    }
}
```

i. builderAgentName: Name of the builder agent

This event is emitted by the supplier to know which builder agent is in the workshop and is available to receive items.

E_ItemPrepared

```
event E_ItemPrepared {
    var workShopName : String
    var entityName : String

    new(workShopName : String, entityName : String) {
        this.workShopName = workShopName
        this.entityName = entityName
    }
}
```

ii. workShopName: The name of the workshop from where the item has to be collected.
iii. entityName: The entity from which the item has to be collected.

This event is emitted by the builder agent to notify the deliverer agent that the item has been assembled completely and is ready to be stored in the storage area.

Behaviour

ChargeAgent

The ChargeAgent behaviour is used by all the three agents, supplier deliverer and builder and is triggered whenever the charge of any entity is below the 50% threshold. After this, any ongoing process for that entity is put on a halt and a nearby charging station is searched to send the entity there for the charging process.

Navigation

After the entity is fully charged, the previously halted process is resumed again by using this behaviour. It stores the last destination of the entity and based on that we can send the entity again to that location.

Skills

KB_AgtCity

This skill is used to store all the functions which are required to interact with the prolog.

Work-flow

The following steps are used to collect all the required items and deliver them to the workshop:

Job Selection

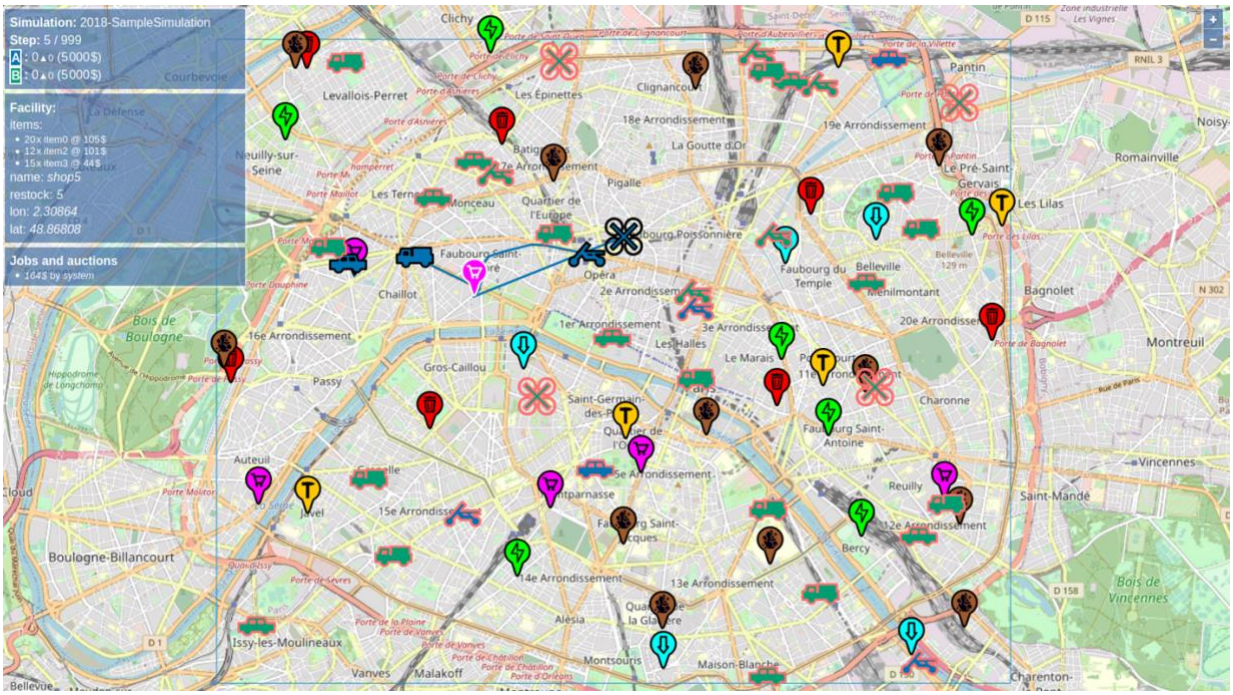
The supplier agent handles the job selection process. Inside *E_EntitySensed* event in *SupplierAgent.sarl*, using prolog function *get_jobs()*, the supplier is selecting all the jobs. To determine whether a job is new or not, we are using a list *jobStatus* to check if it already contains the new *jobID* or not, otherwise process the job and store the *jobID*. For the new job we are extracting all the items required to complete it and storing and for each item, we are finding the closest shop based on Manhattan distance by calling the prolog function *get_shop_for_items()*. We are also finding the nearest entity for the given shop by using function *get_close_entity_to_shop()* inside prolog and assigning the entity to the shop by calling *kb_setEntityToShop()* to store details such as entity name, shop name, item to be purchased and its total count by creating a new predicate *entityToShop(EntityName, Shop, Item, Quant)*. Here the item count represents the total base item count even the base items. required to assemble complex items are also added. We are also using a Hashtable to store the entity and the destination (which is the shop name here). A *goto* action is called to send the entity at the shop.

```
?- get_jobs(EntityName, JobID, Storage, Reward, Start, End, ItemList).
EntityName = entityA28,
JobID = job0,
Storage = storage3,
Reward = 164,
Start = 1,
End = 51,
ItemList = [item4, item5, item6, item7, item8] .
```

```
?- get_shop_for_items(EntityName, ShopCell, ItemCount).
EntityName = entityA28,
ShopCell = [shop(item3, shop5, 44, 14), shop(item2, shop5, 101, 12), shop(item1, shop4, 43, 16), shop(item0, shop5, 105, 11)],
ItemCount = [count(item3, 40), count(item2, 40), count(item1, 40), count(item0, 46)] .
```

```
?- get_close_entity_to_shop('shop4', EntityName).
EntityName = entityA16.
```

```
def kb_setEntityToShop(entityName : String, shopName : String, itemName: String, remainItemQuant : int) : void {
    debug("set kb_setEntityToShop")
    assertFirst("entityToShop(?, ?, ?, ?)", entityName, shopName, itemName, remainItemQuant)
}
```



Buying items

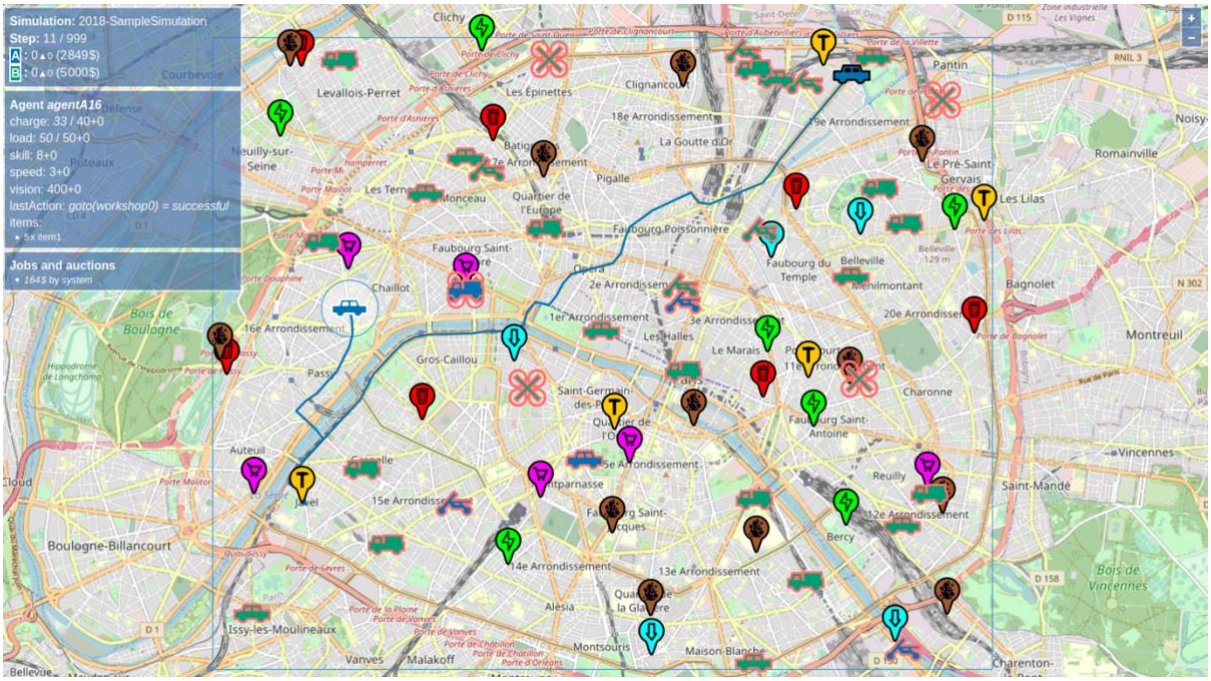
After reaching the shop, from entityToShop predicate, we can extract the items to buy. As the load for each entity is limited, we are finding the max quantity of item it can purchase without overloading itself. After the item is bought, we are decreasing the item count from the entityToShop predicate by calling the function *reduceGetItem*().

```
def reduceGetItem(entityName : String) : void {
    var itemName : String = kb_getItemToBuy(entityName)
    val entity : EntityData = MT_getEntityState(entityName)
    var shopName : String = kb_closeShopForItem(entityName, itemName)
    var itemVolume : int = kb_getItemVolume(itemName.trim())
    kb_setEntityToShop(entityName, shopName, itemName,
        kb_getTotalBaseItemQaunt(itemName) - entity.getLoad() / itemVolume)
    return
}
```

Going to workshop and calling builders

After purchasing the items, the supplier entities are finding the nearest workshop and calling all the builder agents there emitting event *E_GoingForWorkShop*. As mentioned above, to reduce the delay in between giving and receiving items between supplier and builder entities, it is beneficial that all the builder entities are also moving towards the workshop. The destination inside the table *entityLocTable* and predicate *entityToShop* will be updated to the workshop here.

```
?- get_near_workshop('entityA1', WorkShop).
WorkShop = workshop0.
```



Giving and receiving items

After sensing *E_GoingForWorkShop*, the builder agents will also start moving towards the workshop. To know whether the supplier or builder entity has reached workshop or not, we have function *isAtWorkShop()* in prolog. If the builder entity has reached the workshop, it will notify the supplier by emitting the event *E_ReachedWorkShop*, and accordingly, whichever supplier has reached the workshop will pass on the items to builder agents.

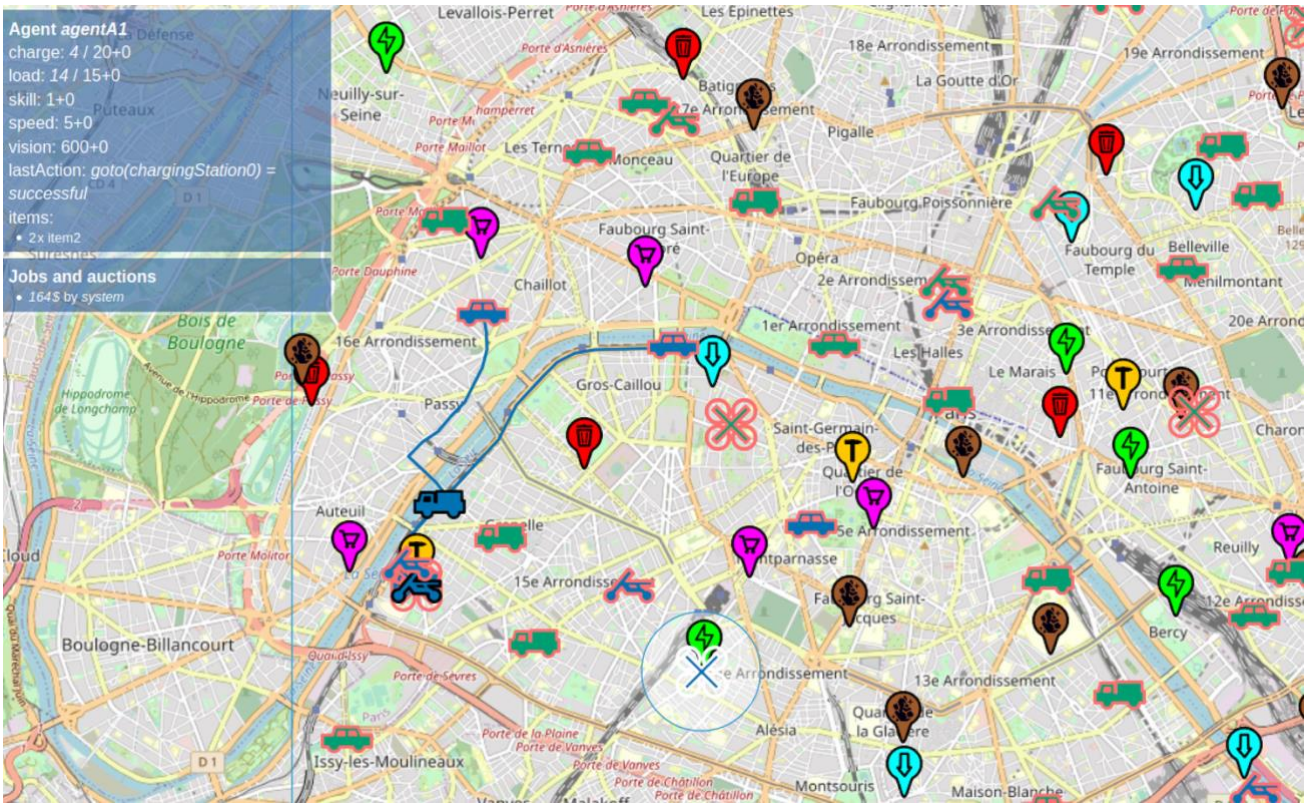
```
isAtWorkShop(EntityName, WorkshopID, Distance):-
    once(percepts_sensed(EntityName, Step, RestData)),
    member(lon(EntityLon), RestData), member(lat(EntityLat), RestData),
    member(workshop(WorkshopID, WorkShopLat, WorkShopLon), RestData),
    get_distance(EntityLat, EntityLon, WorkShopLat, WorkShopLon, Distance), Distance < 0.4.
```

Supplier after giving items

After giving all the items to the builder agents, the supplier entities will again find the nearest shop and repeat the process of buying the remaining items and going to the workshop for passing them to the builder entities.

Charging of entity

If in between the run, the charge is decreased below 50%, then the behaviour ChargeAgent will halt the process and send the entity to the nearest charging station. After charging is done, by using the behaviour Navigation, the entity will determine where it has to go next and resume the halted process.



Strength and Limitations

Strengths

i. The entities are able to collect the items from the workshop without any interruptions.

Limitations

i. When the Supplier and the Builder agent interact, there is a chance of getting *noAction().* This happens during the giving and receiving of items. When the supplier is giving an item and the builder is not able to receive it, and vice-versa.
ii. If the above happens, then the builder will not be able to assemble all the items and hence get stuck in a loop.

References

[1] Pieper, J. Multi-agent programming contest 2017: BusyBeaver team description. *Ann Math Artif Intell* **84,** 17–33 (2018). https://doi.org/10.1007/s10472-018-9589-7