Making RTS games in Casanova

1 Introduction

The video game industry is a very fast growing business, with hundreds of millions of sales per year. Real time strategy (RTS) games are one of the most sold genre [5].

RTS games are the basis of many interesting simulations (not only for entertainment). Indeed, such simulations are of great use for researchers and serious games developers. RTS games are used for AI research [3], simplified military simulations [2], learning [6], etc.

Because of these reasons RTS games are the focus of our research. In particular, we focus our research on how serious games developers and researchers develop RTS games.

Building RTS games, and games in general, is difficult because of complex concurrent patterns on heterogeneous entities, within the constraint that the game must run fast. RTS games are developed by mean of either specialized tools or *general purpose tools* (GPT). Specialized tools are used by companies and usually are privately built-in and maintained inside the company itself [4]. These tools are very expensive to build, so they are typically employed by large studio's (tens of developers). General purpose tools are used by small studio's (roughly up to 10 developers) and are typically open or require licensing to be used. Researchers and serious games developers belong to the small studio category and they use GPT's to implement their research; hence GPT's are the focus of our research.

Among the GPT's we find: Unity3D, Unreal Engine, Game Maker, and ORTS. Unfortunately, such tools are general purpose and thus are not specifically designed with development of RTS games as unique goal. To use GPT's for the building of a specific game genre such as an RTS, since GPT's come with some extension facilities. These extension facilities are called *scripting languages*, which are designed to allow developers to build new behaviors in their tool.

Typically, the scripting language used by a GPL is a *general purpose language* (GPL) such as C# for Unity3D, C++ for Unreal Engine, etc. Unfortunately, GPL's lack specific domain constructs and functionalities related to games. Such limitations affect performance (due to the lack of domain optimization), maintainability, readability and ultimately expressiveness. This leads to poor management of complexities and thus to high costs [**?**].

With big expenses come missed chances: limited resources in research and in the serious games panorama push developers to reduce the amount of features or the depth of these games. Therefore, the opportunity to make innovations is reduced.

Here our work comes into play, tools designed around the domain of games, which do not limit developers in terms of expressiveness and keep development costs low, are necessary in order to (*i*) allow innovative projects to see the end of their development process, and (*ii*) provide developers with the right tools to tackle features that, with limited tools, might not be built.

Our proposal is to investigate how a taxonomy for making RTS games would help serious games developers and researchers to build RTS games. In particular, we investigate to what extent the selected taxonomy is reusable, scalable, flexible, and which can be implemented at high performance. To achieve this we propose a series of implementation strategies within the Casanova 2 language [**?**], a domain specific language (DSL) for games.

Thanks to our approach, RTS's become simpler to build and require less effort. This is all to the advantage of those developers with limited resources that we focus on.

In this paper we discuss the design of RTS's by mean of a case study. We discuss pitfall and difficulties in the making of RTS's and show how are they solved with traditional tools. We introduce our solution a domain language for making games called *Casanova* for the problem of making RTS's. We then evaluate our solution and conclude the paper with the future works. **(*MISSING REFS TO SECTIONS*)**

2 Existing approaches

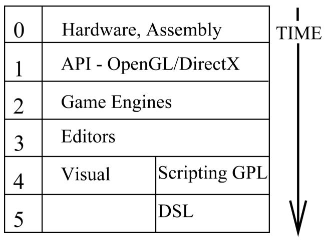

Figure 1: Game development tools evolution

3 PROBLEM - An analysis of RTS's

Implementing an RTS requires writing code for all of the game common elements such as: units, battles, movement, production, resources gathering, statistics, etc.

We identify the common game elements of an RTS by mean of a taxonomy [1]. In this paper the authors introduce a design pattern called *REA* (resource, entity, action) for representing RTS's. In particular the design effectively describes any RTS game in terms of:

- *Resource*, which is any kind of game statistic. A statistic might represent a numerical value of a battle, or the cost to deploy a facility, etc.

- *Entity*, which is any kind of game element that contains resources. We distinguish different entities by mean of their interaction.
- *Action*, which describes an interaction, is used to specialize an entity. Precisely it describes the flow of resources among entities.

Whereas the definition of action given above covers generic type of interaction (like the attack of a ship, or the percentage of construction, etc) a special attention should be given to some kind of actions that are common to all RTS's. We identified these special actions in terms of: *creation*, *deletion*, and *strategy update*:

Creation An entity is crated after some conditions in the game world are met. A condition could be for example the player who decides to create a fleet to attack an enemy player; an automated spawner that after a certain amount of time creates a unit, etc. Furthermore, the creation of an entity typically consumes some game resources of the player. If the resources are not enough then the creation will be postponed or not allowed at all.

Deletion Analogous to creation, an entity is deleted after some conditions in the game are met. A condition could be for example during a battle the life of the entity is lower or equal to zero. Entities removed from the game world are not able to interact with other entities.

Update During the life time of an entity often happens in an RTS that the entity changes its behavior. For example a resource gatherer unity mainly collects resources from all around the world, but if necessary it can also attack; a fleet moving around the world might eventually end up in the local fleets of a planet or take part of a battle. All the just mentioned actions differ from each other, indeed their logics affect different set of resources even though the entity remains the same.

Next, we discuss how the just described model effectively describes an RTS.

4 "Galaxy Wars" case study

Galaxy wars (GW) is an RTS game published in 2012 inspired by the popular board game Risk. Galaxy wars has been used as a case study in related research [?]. The gameplay revolves around your strategic choices, where timing, battles, and resource management are key elements to prevail against the opponents.

The map field is a connected graph where nodes represent planets and links are the physical connection between planets. Planets produce fleets that are controllable by the players. A fleet can either defend its containing planet or move battle against other planets. Fleets move only through links. To every player global statistics are assigned. Global statistics influence fleets attack/defence capabilities as well as fleets speed and planets production. Few special planets feature special ability to update the global statistics of their owners.

4.1 Galaxy Wars with REA

We show an informal idea of implementation of Galaxy Wars with the REA pattern. The elements of Galaxy Wars that follow the REA pattern are: *fleet*, *planet*, *statistic*, and *link*. Resources are *statistics*, the entities are *fleets*, *planets*, and *links*. The possible actions are movement, fight, and upgrade. In GW most of the entities are static. An entity that can be created and deleted is fleet. A fleet is spawned after a player decides to send some units to a planet. A fleet is disposed after either it has reached its destination, or it has lost a battle. Moreover, the fleet entity is the only entity which might change its strategy/behavior during its lifetime (a fleet can either travel along a link or fight in a battle).

In the next section we show how to implement the REA pattern for Galaxy Wars in Casanova 2.

5 SOLUTION - Our idea

Casanova 2 eliminates this separation by allowing rule to be interruptible, thus describing both continuous and discrete dynamics. The rule body is looped continuously, which means that once the evaluation reaches the end, the rule is re-started from the first statement. Writing entity fields is allowed only by using rules. A rule can write an entity field by using a dedicated `yield` statement. Each rule declares a subset of fields called *domain* on which it is allowed to write. Besides each rule has an implicit reference to the world (variable `world`), the current entity (variable `this`), and the time difference between the current frame and the previous (variable `dt`). Casanova 2 supports interruptible control structures and queries, so it natively supports REA.

6 Galaxy Wars in Casanova 2

We now show that Casanova 2 is able to express the design of Galaxy Wars in terms of REA. In what follows the type $[T]$ denotes a list of objects of type T, according to Casanova 2 syntax. We begin by defining the structure of the world entity. The world contains the collection of `Planets` in the map, the collection of `Links` connecting the planets, the collection of `Players`, and a `Controller` that manages the input controller and provides facilities like: the current selected planet, whether a mouse button is down, etc.

```
worldEntity GalaxyWars =
  Planets    : [Planet]
  Links      : [Link]
  Players    : [Player]
  Controller : Controller

  //rules
```

6.1 Resources

The resources are all those elements that influence the game dynamics. In Galaxy Wars the resources are:

- the players statistics (attack, defense, production, research)
- the planets statistics
- the fleets statistics
- the fleets stationed in a planet
- the fleets moving around the map

We use the below properties to model the statistics of the entities: player, planet, and fleet. We use these statistics to amplify or reduce the amount of resources to transfer, thus to alter the impact of the effects of the entity container.

```
entity GameStatistics =
  Attack              : float32
  Defence             : float32
  Production          : float32
  Research            : float32
```

6.2 Entities

Entities represent the resource containers in Galaxy Wars. The entities in Galaxy Wars are:

Planet, which represents the container of stationed fleets. Each planet has its own statistics. Statistics that affect: the attack and velocity of outgoing fleets; the local production of fleets; and the defense capabilities

Link, which is a directed connection between two planets. Links are used by fleets to move around the map

Fleet, which represents the armies of a player. A fleet is made up by ships, and to every fleet statistics are assigned. A fleet might behave differently depending on its current task. Reason why we distinguish different kind of fleet entity, each of which inherits a fleet and is able to accomplish a specific task. The kind of fleets that we identified are:

- **AttackingFleet**, which is a fleet capable to carry out fighting tasks
- **AttackingFleetToMerge**, which represents a special attacking fleet which has just conquered a planet and thus has to be added to the planet stationary fleets (together with the other allied attacking fleets who that attacking the planet)
- **TravelingFleet**, which represents a fleet traveling along a link
- **LandingFleet**, which represents a special traveling fleet which is about to land on the destination planet

Battle, which carries out the fighting task on a planet

Player, which is the owner of entities in game. Every player belongs to a faction. Factions differ among each other based on their statistics. During the game statistics of a player can be changed by means of upgrades.

6.3 Fields

In addition to the resources defined above, additional data fields are used in every entity to support the internal logic of each entity. In what follows we go through each entity and for each entity a code listing the fields and brief description is provided.

Planet Each planet got: its own statistics, the amount of stationed fleets, the incoming fleets, an owner, a link to a possible battle, and the landing fleets.

```
entity Planet =
  Statistics    : GameStatistic
  LocalFleets   : int
  InboundFleets : [Fleet]
  ref Owner     : Option<Player>
  Battle        : Option<Battle>
  LandingFleets : [LandingFleet]
```

Link Besides its source and destination, a link made up of a collection of traveling fleets.

```
entity Link =
  ref Source        : Planet
  ref Destination   : Planet
  TravellingFleets  : [TravellingFleet]
```

Fleet A `Fleet` is made of: statistics, the amount of ships, a ref to the link on which its is traveling, an owner, a destroyed flag, and the position.

```
entity Fleet =
  Statistics  : GameStatistic
  Ships       : int
  ref Link    : Link
  ref Owner   : Player
  Destroyed   : bool
  Position    : Vector3
```

AttackingFleet An attacking fleet is a specialized fleet that contains: a ref to the actual fleet and a reference to its battle.

```
entity AttackingFleet =
  ref MyFleet : Fleet
  ref MyBattle : Battle
```

AttackingFleetToMerge An attacking fleet to merge is a specialized fleet that contains: a ref to the actual fleet and a reference to the attacking fleet with which it has to join.

```
entity AttackingFleetToMerge =
  ref MyFleet : Fleet
  ref FleetToMergeWith : AttackingFleet
```

TravelingFleet Is a specialized fleet that contains: a reference to the actual fleet, the destination planet, and the velocity.

```
entity TravelingFleet =
  MyFleet : Fleet
  ref Destination : Planet
  Velocity : Vector3
```

LandingFleet Contains the reference to the actual fleet.

```
entity LandingFleet =
  MyFleet : Fleet
```

Battle A battle is made up of:the planet where the battle is hosted, a collection of attacking fleets, the possible amount of losses of the hosting planet, the possible amount of losses of the attacking fleets, the just destroyed attacking fleets, and the just arrived attacking fleets that have to be grouped into the attacking fleets.

```
entity Battle =
  ref MySource     : Planet
  AttackingFleets : [AttackingFleet]
  DefenceLost      : Option<int>
  AttackLost       : Option<int>
  FleetsToDestroyNextTurn  : [AttackingFleet]
  FleetsToMerge    : [AttackingFleetToMerge]
```

Player A player is made of the statistics of its faction and the its name.

```
entity Player =
  Statistics : GameStatistic
  Name : string
```

6.4 Actions

Actions are the only way, according to REA, to exchange resources like the amount of attacks in a battle, the amount of fleets to produce, etc. In Galaxy Wars we identified three kind of actions: battle, production, and upgrade.

6.4.1 Battle

A Battle action involves a planet `MySource` and a series of `AttackingFleets`.

Attack In this design only one selected attacking fleet at a time can attack `MySource`, precisely the fleet which is in the head of the `AttackingFleets` collection. Every random time an amount of damage is computed and stored in the `Battle` entity. Before computing the amount of damage, we check that there are still fleets in the `AttackingFleets` collection.

```
entity Battle =
  ...
  rule AttackLost, DefenceLost =
    yield None, None
    wait 1.0f
    if AttackingFleets.Count > 0 then
        yield // amount of looses based on the
              // statistics of both the attacking
              // fleet and the planet
```

The amount of damage represents the damage that has be applied to both: the selected attacking fleet and the defending planet. This damage will always be applied since every instance of `AttackingFleet` and `Planet` involved in a battle keep updating their amount of fleets.

```
entity AttackingFleet =
  ...
  rule MyFleet.Ships =
    wait MyBattle.AttackLost.IsSome && MyBattle.AttackingFleets.
        Head = this
    yield MyFleet.Ships - MyBattle.AttackLost

entity Planet =
  ...
  rule LocalFleets =
    wait Battle.IsSome && Battle.DefenceLost.IsSome
    yield LocalFleets - Battle.DefenceLost.Value
```

Attacking fleet selection A random circularly selection based on random time is used to allow all attacking fleets to attack the planet.

```
entity Battle =
  ...
  rule AttackingFleets =
    .| AttackingFleets.Count <= 1 => yield AttackingFleets
    .| _ =>
      wait Random.Range(1.0f, 2.0f)
      yield AttackingFleets.Tail @ [AttackingFleets.Head]
```

Ownership We change the owner of a planet when at the end of a battle the attacker list is not empty. When we change the owner we also update the amount of `LocalFleet`, by summing all the fleets that share the same new owner and that are attacking the planet.

```
entity Planet =
  ...
  rule Owner, LocalFleets =
    if Battle.IsSome &&
       LocalFleets = 0 &&
       Battle.AttackingFleets.Count > 0 then
    let new_owner = Battle.AttackingFleets.Head.MyFleet.Owner
    let fleets_to_add =
          Battle.AttackingFleets
          .Where(f => f.MyFleet.Owner = new_owner && f.MyFleet.
              Ships > 0)
          .sum(f => f.MyFleet.Ships)
    yield Some new_owner, fleets_to_add
```

6.4.2 Production

The spawning of a new fleet follows the following schema: if a battle is ongoing on a planet then the production is interrupted and the planet keeps polling the battle in order to update its local fleets; if the planet is neutral (it is not possessed by any player) then production is interrupted; eventually if the planet is not neutral and the is not an ongoing battle then we wait some time, which depends on the production statistics of both the player and the planet, and then we add a new fleet to the local fleets.

```
entity Planet =
  ...
  rule Owner, LocalFleets =
    .| Battle.IsSome => yield LocalFleets
    .| Owner.IsNone => yield 0
    .| _ =>
      wait //time depending on the owner
           //and the planet production
      yield LocalFleets + 1
```

6.4.3 Upgrade

When the planet is selected and a key associated to an upgrade is down we: (*i*) wait an amount of time (which depends from the owner and the planet research statistics), and then (*ii*) we upgrade the selected statistic. If the planet is neutral less then its statistics are, by default, set to `1`.

```
entity Planet =
  ...
  rule Statistics.STAT =
    .| Owner.IsNone -> yield 1
    .| _ ->
      wait IsSelected && KeyPressed(STAT_KEY)
      wait //time depending on the owner
           //and the planet research
      yield max(MAX_STAT, Statistics.STAT + 1)
```

6.4.4 Creation

In Galaxy Wars we create entities when: (*i*) a battle is about to start, and (*ii*) when a fleet is spawned.

Battle On a planet a battle is created either when either the planet is neutral and a fleet is approaching the planet; or the planet is not neutral, there are no battles ongoing on the planet, and an enemy fleet is approaching the planet.

```
entity Planet =
  ...
  rule Battle =
        let exits_an_enemy_fleet =
          LandingFleets.Count = InboundFleets.Count |> not
    if (Owner.IsNone && Battle.IsNone && exits_an_enemy_fleet) ||
       (Owner.IsSome && exits_an_enemy_fleet) then
      yield Some (new Battle(this))
      wait Battle.AttackingFleets.Count <= 0
    yield None
```

Fleet We consume all local fleets of a planet and move them through the link when the source planet is selected (and its fleets are greater than 0) and the destination planet is selected as well. The selection logic of planets is in the planet entity and it is not covered in this presentation.

```
entity Link =
  ...
  rule TravellingFleets, Source.LocalFleets =
    wait Source.Selected && Destination.RightSelected &&
         Source.Owner.IsSome && Source.Battle.IsNone &&
         Source.LocalFleets > 0
    yield new TravellingFleet(Destination) :: TravellingFleets, 0
```

6.5 Deletion

Analogously to creation, in GW the entities which might be disposed during a game are battles and fleets.

Battle The logic of the deletion of a battle is tightly related to the logic of its creation. In the previous subsection a battle is disposed only when the amount of `AttackingFleets` is equals to `0`.

Fleet The general logic of the deletion of a fleet follows the following criteria: if the fleet got no ships then it has to be destroyed. In code, a fleet destroys itself when the amount of its `Ships` is less or equals to zero.

```
entity Fleet =
  ...
  rule Destroyed =
    wait Ships <= 0
    yield true
```

Special attention goes to when the ship is: fighting or about to land or traveling.

Fighting If during a battle the attacker manages to conquer the planet then all the attacking fleets that share the same owner of the just conquered planet have to be destroyed.

```
entity AttackingFleet =
  ...
  rule MyFleet.Destroyed =
    wait (MyBattle.MySource.Owner.IsSome &&
          MyFleet.Owner = MyBattle.MySource.Owner)
    yield true
```

And filtered from the attacking fleets collection and moved into. We move such fleets to destroy in a different collection so their logic will not affect the logic of the battle. Entities in such collection last for one frame.

```
entity Battle =
  ...
  rule FleetsToDestroyNextTurn =
    yield
      [for f in AttackingFleets do
       where (MySource.Owner.IsSome && f.MyFleet.Owner = MySource.
           Owner))
       select f]
```

Fleets that have not managed to conquer the planet, but which have been destroyed are filtered from the attacking list collection.

```
entity Battle =
  ...
  rule AttackingFleets =
    yield
      [for f in AttackingFleets do
       where (not f.MyFleet.Destroyed) &&
             not FleetsToDestroyNextTurn.Contains(f)
       select f]
```

Landing A landing fleet is a traveling fleet, which is about to land and whose owner is the same as its destination planet. In order to not to add twice the ships of the landing fleet among the local fleets of the destination planet, a landing fleet lasts one frame in the game (we destroy it at the first frame).

```
entity LandingFleet
  ...
  rule MyFleet.Destroyed = yield true
```

New landing fleets are continuously added to the local fleets.

```
entity Planet
  ...
  rule LocalFleets =
    yield LandingFleets.Sum(f => f.MyFleet.Ships) +
          LocalFleets
```

Traveling When a traveling fleet has reached its destination, the fleet is automatically filtered by the link.

```
entity Link =
  ...
  rule TravellingFleets =
  yield
    [for f in TravellingFleets do
     where Vector3.Distance(f.MyFleet.Destroyed |> not &&
                           f.MyFleet.Position, Destination.
                               Position) > Destination.
                               MinApproachingDist
     select f]
```

6.6 Behavior update

An entity during its life cycle might change its behavior based on its state. An example of this kind of behavior in GalaxWars could be identified in the fleet entity. For example an attacking fleet behaves differently than a moving fleet. In Casanova we distinguish these two cases by mean of two different entities that share some common properties, but implement different rules.

InboundFleet When a fleet, traveling along a link, is approaching the destination, the planet has to choose whether to: (*i*) add the fleet to the planet local fleets (see production of a planet above), or (*ii*) add the fleet to a battle. To implement the just described scenario we start with the definition of a buffer to place in the Planet entity called InboundFleets. The InboundFleets of a planet represents all fleets that are approaching at a specific time the planet.

```
entity Planet =
  ...
  rule InboundFleets =
    yield [for l in world.Links do
           where (l.Destination = this)
           for f in l.TravellingFleets do
           where (Vector3.Distance(f.MyFleet.Position, Position) <=
                MinApproachingDist)
           select f.MyFleet]
```

InboundFleets acts like a dispatcher. When a fleet enters the InboundFleets collection, other entities are able to consume it for their internal logics. To avoid entities to consume twice the same fleet, fleets in InboundFleets last for one frame before being disposed. When an entity consumes an inbound fleet it decides what behaviors to apply to the selected fleet. This is done by assigning the fleet to an other instance, of different type, which contains the fleet but provides new rules.

Attacking fleets to merge Fleets that come from the same link and that share the same owner have to e joined. To do so, every enemy inbound fleet that share the same source link of a fleet stored in AttackingFleets is selected and converted into an AttackingFleetToMerge. Eventually all the fleets of type AttackingFleetToMerge are stored into FleetsToMerge for the duration of one frame.

```
entity Battle =
  ...
  rule FleetsToMerge =
    yield
      [for i_f in MySource.InboundFleets do
       for a_f in AttackingFleets do
       where (not a_f.MyFleet.Destroyed &&
              i_f.Link = a_f.MyFleet.Link)
       select (new AttackingFleetToMerge (i_f, a_f))]
```

When we create an attacking fleet to merge the reference to the actual attacking fleet is stored in the FleetToMergeWith of the attacking fleet to merge. An attacking fleet every frame iterates through all the attacking fleets of its battle and selects those fleets to merge whose FleetToMergeWith is the attacking fleet. After the selection, the amount of ships of the selected attacking fleets to merger is sum and added to the local fleets of the attacking fleet.

```
entity AttackingFleet =
  ...
  rule MyFleet.Ships =
    yield MyBattle.FleetsToMerge
          .Where(f => f.FleetToMergeWith = this)
          .sum(f.MyFleet.Ships)
         + MyFleet.Ships
```

Attacking fleet A battle entity every frame selects the enemy fleets from the inbound fleets of its MySource field and adds them to its AttackingFleets, as long as the selected fleets are not in `FleetsToMerge`. Before adding the inbounding attacking fleets, every inbound enemy fleet is converted to an attacking fleet.

```
entity Battle =
  ...
  rule AttackingFleets =
    yield
      [for f in MySource.InboundFleets do
       let is_ship_to_merge = FleetsToMerge.Contains(f)
       where is_ship_to_merge &&
             (MySource.Owner.IsNone ||
              not (f.Owner = MySource.Owner))
       select new AttackingFleet(f, this)]
```

The moment a fleet becomes an attacking fleet and it is added to the `AttackingFleet` collection the new logic (the one of the attacking) can be run.

Landing fleet A planet every frame selects all allied fleets among its inbound fleets and adds them to the `LandingFleet` collection, so the planet later can add those fleets to its local fleets. Before adding the inbound allied fleets, every fleets is converted to an inbound fleet.

```
entity Planet
  ...
  rule LandingFleets =
    if Owner.IsSome then
      yield
        [for inbound_fleet in InboundFleets do
         where (inbound_fleet.Owner = Owner)
         select (new LandingFleet(inbound_fleet))]
    else yield []
```

7 DETAILS - Abstraction

In this section we show how Casanova 2 can implement the REA pattern, and also extend it with the CDM pattern.

7.1 Resources

Resources can be modeled as a Casanova entity with no rules containing a field for each of the resources used in the REA pattern. In a game we might have different resource entities for different group of resources. We define a resource entity by first defining its name `ResourceName` and then by listing the resources contained in it. A resource in Casanova is a field and it is defined as tuple `Resource * Type` where the first item refers to the field name while the latter refers to the field type. In the following we show a generalized description for a generic resource entity.

```
entity ResourcesName =
  R_1 : T_1
  R_2 : T_2
  ...
  R_n : T_n
```

7.2 Entities

REA entities can be modeled directly as Casanova entities. To avoid possible confusion arising from the ambiguity between the `entity` keyword in Casanova and entities in the REA pattern from now on we will refer to the latter as *actors*. An actor will contain the `Resources` (of type ResourcesName) and a series of rules that will act as the constant, mutable, and threshold actions of REA.

```
entity Actor =
  Resources              : ResourcesName
  //constant actions
  //mutable actions
  //threshold actions
```

7.3 Actions

Following the REA pattern, we divide the actions into 3 categories: (*i*) Constant transfer, (*ii*) mutable transfer, and (*iii*) threshold transfer. We model the REA transfers as rules in Casanova.

An action in REA simply puts in communication a source with a target. In Casanova the source is the action/rule container while the target is an entity containing a field that refers to the source. The target checks the source reference whenever it needs to interact with it. Precisely, the source resources are read by the target actor periodically to locally update their fields[1].
The resources to transfer generated by actions are stored inside the source entity and precisely inside its resource entity. We refer to the resources to transfer as `Transfers` in Casanova.
The definition of the actions will be shown in the next three paragraphs.

Constant transfer A constant transfer simply sums the resources of a source entity to the resources of a target. The following rule, which is contained in the source entity, updates the `Transfers` whenever a condition is met (we assume that `restrictions` is a predicate that specifies a condition to apply the action). The rule waits one frame, to ensure that the target actor reads the change, before resetting the `Transfers`.

```
enity SourceActor =
  Resources   : ResourcesName
  ...
  rule Resources.Transfers =
    wait restrictions
    yield Some(some_resources)
    yield None
```

Every time some `Transfers` are produced the target actor reads them and updates its resources accordingly. We use the same `restriction` as in the source entity to ensure that the generated `Transfers` belong to that specific target instance. We assume for brevity that we have a + operator for the entity `Resources`, which behaves like a vector sum, to be used by the aggregate function `sum` in the query.

[1]The action rules do not modify the target actor directly to grant encapsulation, unless the target is the source itself. An extensive discussion about encapsulation in games, as well as an optimizer for encapsulated code in Casanova can be found in [?]

```
enity TargetActor =
  Resources   : ResourcesName
  ref ASource : SourceActor
  ...
  rule Resources =
    wait ASource.Transfers.IsSome & restrictions
    yield Resources + ASource.Transfers.Value
```

Mutable transfer In the mutable transfer the resources are moved from the source to the targets. A transfer can be also negative in REA. In case of negative transfers we simply swap the logic so the source implement the behavior of the target and vice-versa. The rule of the mutable transfer behaves almost the same as for the continuous transfer. Only difference is that in the source together with setting the `Transfers` by an amount `some_resources` we also remove the same `some_resources` from the source resources. Again, we assume for brevity that we have a `-` operator for the entity `Resources`, which behaves like a vector difference, to be used by the aggregate function `diff` in the query.

```
enity SourceActor =
  Resources   : ResourcesName
  ...
  rule Resources.Transfers, Resources\{Transfers} =
    wait restrictions
    yield Some(some_resources), Resources\{Transfers} - some_resource
    yield None
```

Threshold transfer The threshold transfer is a constant or a mutable transfer that executes the resources transfer, as in the examples above, until a certain `threshold_condition` is satisfied. Once we meet the `threshold_condition` a series of output values are yielded and then reseted. For this kind of action we need to extend the source entity definition with additional fields to store the output of the rule.

```
enity SourceActor =
  Resources   : ResourcesName

  Output_0 : Option<T_0>
  Output_1 : Option<T_1>
  ...
  Output_n : Option<T_n>

  ...
  rule Resources.Transfers, Output_0, ..., Output_n =
    .| threshold_condition ->
      yield None, Some value_0, ..., Some value_n
      yield None, None, ..., None

    .| _ ->
      wait restrictions
      yield Some(some_resources), Output_0, ..., Output_n
      yield None, Output_0, ..., Output_n
```

7.4 Creation

Creation of an entity follows always an event. In Casanova we can combine the creation expression with any action. This is allowed since inside a rule in Casanova statements are run imperatively. The following shows a generalization for the creation of an object after an action is run. An entity of type `SomeEntity` is spawned after an action is run.

```
enity SourceActor =
  SomeObject : SomeEnity
  ...
  rule Resources.Transfers, SomeObject =
    // an action
    yield Resources.Transfers, new SomeEnity(some_parameters)
```

7.5 Deletion

Deletion follows the same code of creation, but we must take into consideration the following: if an instance `O` is about to get destroyed, all instances `Is` that share some logic with `O` must be notified that `O` is about to get destroyed. An instance of `Is` knows that `O` is about to get destroyed when `O` is moved into a special field called `DestroyedO`. `O` is moved into `DestroyedO` for a certain amount of time before we reset the `DestroyedO` field. In the following code `SourceActor` contains, besides the usual fields, also a reference to an object `O` of type `Object` and the `DestroyedO`, which is an option of type `Object`.

```
enity SourceActor =
  ref DestroyedO : Option<Object>
  O : Option<Object>
  ...
  rule O, DestroyedO =
    wait restrictions
    let acc = O
    yield None, Some acc
```

7.6 Change of strategy

An entity moves according to some logic. In this case we can apply a constant transfer to for example update an entity position according to its velocity. An entity might change its behavior according to some conditions. For example a fleet might change from traveling to attacking. This kind of behavior might resemble the strategy pattern and in Casanova we implement it by explicitly moving the moving object from a container of type `F` into an other container of type `T`. `T` and `F` share few information like physical information, graphics, etc. but differ in terms of behavior. We can generalize the movement behavior by combining the above actions. We start first with the definition of an entity `MovingActor` which is an entity that moves the position of its instance according to its velocity.

```
enity MovingActor =
        Resources : ResourcesName

        rule Resources.Position = yield Resources.Position +
            Resources.Velocity * dt
        //.. other rules and fields
```

An entity `ActionActor` is an entity that shares some structure with the `MovingActor` entity (for example the position or the velocity) but implements different rules.

```
enity ActionActor =
  Resources : ResourcesName

  rule Resources.Position = // move around a target for example
  rule Resources.Life = // remove life if the entity is hit
  //.. other rules and fields
```

A `SourceActor` is an entity that contains among its fields a `MovingActor` and an `ActionActor` field. `SourceActor` combines the actions described above so that when an entity of type `MovingActor` needs to behave like an `ActionActor` we use the deletion pattern to move the `MovingActor` into a temporary location, so to give time notify all the entities, and then we assign it to `ActionActor`. The code below shows the just presented solution.

```
enity SourceActor =
  AActor : Option<ActionActor>
  ref AActorToDestroy : Option<ActionActor>
  MActor : Option<MovingActor>

  rule AActor, AActorToDestroy = // same logic as deletion

  rule MActor =
    wait AActorToDestroy.IsSome
    yield new MActor(AActorToDestroy.Value.Resources) |> Some
```

8 Evaluation

CNV vs C# vs Python

9 Conclusions

References

[1] Mohamed Abbadi, Francesco Di Giacomo, Renzo Orsini, Aske Plaat, Pieter Spronck, and Giuseppe Maggiore. Resource entity action: A generalized design pattern for rts games. In *Computers and Games*, pages 244–256. Springer, 2014.

[2] Michael Buro. Real-time strategy games: A new ai research challenge. In *IJCAI*, pages 1534–1535, 2003.

[3] Michael Buro. Call for ai research in rts games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 139–142, 2004.

[4] Michael Buro and Timothy Furtak. On the development of a free rts game engine. In *GameOn'NA Conference*, pages 23–27. Citeseer, 2005.

[5] Mark Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005.

[6] Manu Sharma, Michael P Holmes, Juan Carlos Santamaría, Arya Irani, Charles Lee Isbell Jr, and Ashwin Ram. Transfer learning in real-time strategy games using hybrid cbr/rl. In *IJCAI*, volume 7, pages 1041–1046, 2007.