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 [6].

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 [4], simplified military simulations [3], learning [7], 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 [5]. 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 describe how to implement RTS games, based on some taxonomy of such games, in a way that is reusable, scalable, flexible, and shich 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

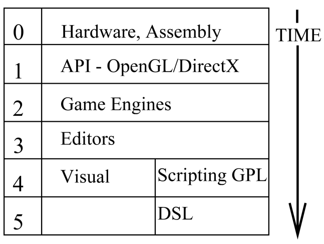

Figure 1: Game development tools evolution

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

Analogously 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.

Strategy 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.

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. 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. The only entity that can be created and deleted are fleets. 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 is an iteration of Casanova, a DSL for game development. A Casanova program is a tree of data structures called *entities*. The root of the tree is called *world*. Each entity contains a set of *rules* which modifies the fields and describe continuous dynamics. Discrete dynamics (the dynamics of the game that require timing or synchronization conditions) are expressed by coroutines implemented with a variation of the state monad. 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 REACDM 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:

- amount of fleets in a Planet
- the player statistics (attack, defence, production, research)
- planet and the ships statistics
- the amount of travelling ships in a link

We use the following properties to model the statistics of the entities: player, planet, and fleet.

```
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: `Planets`, `Fleets`, `Players`, and `Links`. A player represents the commander managing all the elements of the RTS. It contains starting statistics that define a faction (a player might start with more production but less research). During the game statistics can be increased with upgrades.

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

A planet represents the container of stationed ships. Each planet has its own statistics. The attack capabilities of a fleet can be increased depending on the attack value of the source planet. Research affects the velocity of the outgoing fleet. Defense and production affects the ship construction speed and defence capabilities (used when a planet is attacked). We use inbound fleets to select the incoming fleets from a link that is targeting the current planet. `Owner` is an option because not all planets are controlled by a player (in the beginning every player owns at most one planet and the other are neutral).

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


  //rules
```

A `Fleet` is defined by an `Owner`, the amount of ships, and its own statistics. Attack and defence are used in combat. A fleet has no production (it is always set to 0), while the research defines its speed.

```
entity Fleet =
  Statistics          : GameStatistics
  Ships               : int
  ref Owner           : Player
```

```
  //rules
```

Battles in GalaxyWars are carried out by mean of entities of type `Battle`. A battle is made up by:

- `MySource` the planet where the battle is hosted,
- `AttackingFleets` the fleets that are waiting to attack `MySource`,
- `SelectedAttackingFleet` the fleet that is attacking `MySource`,
- `DefenceLost` the amount of fleets lost after a match by `MySource`, and
- `AttackLost` the amount of fleets lost after a match by `SelectedAttackingFleet`.

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

  //rules
```

A link is a directed connection between two planets. Besides its `Source` and `Destination`, a link also contains `TravellingFleets` that is collection of ships that are currently traveling along the link.

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

  //rules
```

Eventually, we introduce two entities `TravelingFleet` and `AttackingFleet`. The former represent a fleet that is able to move around the map; the latter represents a ship able to carry out fighting tasks. A traveling fleet and an attacking fleet inherit the same `Fleet` entity. The reason is because an attacking ship is a traveling ship (same model, same position, same amount of fleets, etc.) that implements different set of behaviors. Furthermore, an attacking fleet contains also a reference its battle.

```
entity AttackingFleet =
  inherit Fleet
  ref MyBattle : Battle
  //rules

entity TravelingFleet =
  inherit Fleet
  ref Destination : Planet
  //rules
```

6.3 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. Input actions are left out of our explanation, because the transfer behavior is not completely under the control of Casanova (input rely on external facilities/libraries). However, capturing user input in Casanova 2 is possible; some examples can be found on `https://github.com/vs-team/casanova-mk2/wiki/Casanova-Samples-and-Demos-Tutorials`.

A **Battle** action involves a planet `MySource` and a series of `AttackingFleets`. In the following code The rule carries the attacking fleet selection. Every random time an entity among `AttackingFleets` is selected and stored into `SelectedAttackingFleet`. If in the mean time the battles ends we stop the selection and wait the battle to get disposed. If the selected ship is destroyed and there are `AttakingShips` available then we select an other attacking fleet among the `AttakingShips`.

```
entity Battle =
  //fields

  rule SelectedAttackingFleet, AttackingFleets =
    .| SelectedAttackingFleet.Fleet.Destroyed && AttackingFleets =
        [] ->
       yield None, []
    .| SelectedAttackingFleet.Fleet.Destroyed ->
       let new_selected_fleet = AttackingFleets.[random(0,
           AttackingFleets.Count - 1)]
       yield new_selected_fleet, new_attacking_fleets -
           new_selected_fleet
    .| _ ->
       wait random
       let new_selected_fleet = AttackingFleets.[random(0,
           AttackingFleets.Count - 1)]
       yield new_selected_fleet, new_attacking_fleets +
           SelectedAttackingFleet
  ...
```

An other rule computes the amount of damage to apply every random amount of time to both `SelectedAttackingFleet` and `MySource`.

```
  ...
  rule DefenceLost, AttackLost =
    yield None, None
    wait random
    //code that computes the damage to apply to both
    //the attacker and defender based on their attack/defence
    //stats and on the mount of attacking fleets
```

Every instance of `AttackingFleet` and `Planet` involved in a battle keep updating their amount of fleets, so to carry our their internal logic.

```
entity AttackingFleet =
  //fields

  rule Destroyed =  wait Ships = 0; yield True
  rule Ships =
    wait MyBattle.AttackLost.IsSome && MyBattle.
        SelectedAttackingFleet = this
    yield Ships - MyBattle.AttackLost.Value

entity Planet =
  //fields and rules

  rule LocalFleets =
```

```
    wait Battle.DefenceLost.IsSome
    yield LocalFleets - Battle.DefenceLost.Value
```

We change the owner of a planet only in two possible ways: (i) after the end of a battle and the attacker still got some armies alive, first rule (we set the current owner to `None`, after the victory of an attacking fleet, in order to reset the internal logics of the planet associated to the previous owner), and (ii) when the planet is neutral and there are fleets that are inbounding, second rule. In the last case case the first fleet in `InboundFleets` gets the ownership.

```
entity Planet =
  //fields and rules

  rule Owner =
    wait Battle.IsSome && LocalFleets = 0 &&
    Battle.SelectedAttackingFleet.IsSome &&
    Battle.SelectedAttackingFleet.Value.Ships > 0
    yield None
    yield SelectedAttackingFleet.Value

  rule Owner, InboundFleets, LocalFleets =
    wait Owner.IsNone && Battle.IsNone &&
         InboundFleets.Count > 0
    let selected_fleet = InboundFleets.Head
    yield selected_fleet.Owner,
          InboundFleets - selected_fleet
          selected_fleet.Fleets
```

A **production** action involves a `Planet` entity and the owner production statistics. The action is performed by a rule that waits an amount of time, which depends on the production statistics of both planet and owner, and then it adds a fleet to `LocalFleets`. If the planet changes owner (for a frame `Owner` is `None`) we reset the rule behavior and set `LocalFleets` to `0`.

```
entity Planet =
  //other field and rules

  rules LocalFleets =
    .| Owner.IsNone -> yield 0
    .| _ ->
      wait Statistics.Production * Owner.Value.Production
      yield LocalFleets + 1
```

Upgrading the stats of a planet is performed by waiting the planet to get selected. 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), then (ii) we upgrade the selected statistic. If the planet is `Owner` less then its statistics are, by default, set to `1`.

```
entity Planet =
  //other field and rules

  rule Statistics.STAT =
    .| Owner.IsNone -> yield 1
    .| _ ->
      wait world.SelectedPlanet = this &&
           KeyPressed(STAT_KEY)
      wait Owner.Value.Research * Statistics.Research
      yield max(MAX_STAT, Statistics.STAT + 1)
```

6.3.1 Creation

In Galaxy Wars we create entities when: (*i*) a battle is about to start, and (*ii*) when a fleet is spawned.
Given a planet `P`, a battle is created created if there are no other battles in progress in `P`, `P` is owned by a player, and `InboundFleets` contains at least an enemy fleet. A battle is disposed when the planet has lost its owner (`wait Owner.IsNone`) or when there are no `AttackingFleets` left.

```
entity Planet =
  //fields and rules

  rule Battle =
    wait Battle.IsNone &&
         Owner.IsSome &&
         InboundFleets.Contains(f => f.Owner <> Owner.Value)
    yield new Battle(this)
    wait Owner.IsNone
    yield None
```

A fleet is sent though a link only if the source planet contains enough local fleets and there are no battles in progress. The local fleets of the selected planet are set to `0` when the user decides to send fleets through a link.

```
entity Link =
  //fields and rules

  rule TravellingFleets, Source.LocalFleets =
    wait world.SelectedPlanet = Source &&
         world.DestinationSelectedPlanet = Destination &&
         world.Battle.IsNone
         world.Source.LocalFleets > 0
    new TravelingFleet(new Fleet(Source, world.Source.LocalFleets))
         @
        TravellingFleets, 0
```

6.4 Deletion

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

The logic of the deletion of a battle is tightly related to the logic of its creation. In the code above a battle is disposed only when the `Owner` is `None`. This means that the planet owner has lost its owner, hence the battle is over.

The logic of the deletion of a fleet depends on whether the fleet is fighting or traveling. When fighting a fleet is destroyed when its life is below or equal to zero. When the life is below or equal to zero the fleet field `Destroyed` is set to `true`.

```
entity AttackingFleet =
  inherit Fleet
  ref MyBattle : Battle
  rule Destroyed =
    wait Life <= 0.0f
    yield true
```

This is necessary in order to notify other entities that the fleet has been destroyed and to allow the battle to remove it from its `AttakingFleets`.

```
entity Battle =
  //fields and rules

  AttackingFleets : [Fleet]
  rule AttackingFleets =
    yield [for fleet in AttackingFleets do
           where not fleet.Destroyed
           select fleet]
```

When traveling, a fleet is destroyed upon it has reached its destination. In this case, when the fleet is among the `InboundFleets` of its `Destination` the field `Destroyed` of the fleet is set to `true`. An additional check is added before destroying the fleet. If the owner has changed or there is a battle running on the planet then the fleet should not be destroyed, since the fleet will turn into an attacking fleet.

```
entity TravelingFleet =
  inherit Fleet
  ref Destination : Planet
  rule Destroyed =
    wait self in Destination.InboundFleets &&
         Destination.Battle.IsNone
    yield True
```

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

```
entity Link =
  //fields
  rule TravellingFleets =
    yield [for f in TravellingFleets do
           where Vector3.Distance(f.Position, Destination.Position)
                 > MIN_DIST
           select f]
```

6.4.1 Change of strategy

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 with the fleet entity. For example an attacking fleets 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.

When a traveling fleet is approaching the planet at the end of the link, the planet has to choose whether to: (*i*) add the fleet in the planet local fleets, (*ii*) add the fleet to a battle, (*iii*) or forward the fleet to an other link. 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 `X` represents all fleets that are approaching at a specific moment the planet `X`.

```
entity Planet =
  InboundFleets       : [Fleet]

  ..//other fields and rules

  rule InboundFleets =
    yield [for l in world.Links do
```

```
            where l.Target = this &&
            for f in l.Fleets do
            where Vector3.Distance(f.Position, this.Position) <
                MIN_DIST
            select f.]
```

InboundFleets acts like a dispatcher. Entities are notified about change state of a fleet the moment it enters enters the InboudFleets. 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 fleets it decides how to change the fleet behavior.
A battle entity every frame selects the enemy fleets from the inbound fleets of its MySource field and adds them to its AttackingFleets. Before adding the inbounding enemy fleets to the AttackingFleets collection, every inbound enemy fleet is converted to an attacking fleet.

```
entity Battle =
  AttackingFleets : [AttackingFleet]
  rule AttackingFleets =
    if MySource.Owner.IsSome then
      yield [for f in MySource.InboundFleets do
             where f.Owner <> MySource.Owner.Value
             select new AttackingFleet(f)] @ AttackingFleets
```

A link forwards a fleet F to its own link when F is inside Source.InboundFleets and F.NextTarget is the link Destination.

```
entity Link =
  ref Source            : Planet
  ref Destination       : Planet
  ref TravellingFleets  : [TravelingFleet]
  //... other rules
  rule TravelingFleets =
    wait Source.Battle.IsNone
    yield [for f in Source.InboundFleets do
            where f.NextTarget = Destination &&
            select new TravelingFleet(f)] @ TravellingFleets
```

Eventually, a fleet is added to the local fleets of a planet if the fleet is in the InboundFleets collection and it shares the same owner of the planet.

```
entity Planet =
  LocalFleets   : int
  ref Owner     : Option<Player>
  InboundFleets : [InboundFleet]
  ...//other field and rules

  rules LocalFleets =
    wait Owner.IsSome
    yield LocalFleets +
           [for f in InboundFleets do
            where Owner.Value = f.Owner
            select f
            sum]
```

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 =
  R1 : T1
  R2 : T2
  ...
  Rn : Tn
```

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. Since very often actions are run only after a predicate is satisfied and the predicate might depends on the target properties, among the source fields a field that refers to the target is added. The target checks the source reference whenever it needs to interact with it.
A generalization for this approach could be the following: the source contains `S` a reference to a target `T`. Whenever the `T` needs to interact with an action it traverses the world to find an `S` that is targeting `T`. For practicality reasons from now on we use the not generalized version.

The action rules do not modify the target actor directly to grant encapsulation, unless the target is the source itself. 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 the 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
  ref TSource :TargetActor
  ...
  rule Resources.Transfers =
    wait restrictions
    yield Some(some_resources)
    yield None
```

Every time some `Transfers` are produced the target actor reads them and update 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.

```
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
  ref TSource :TargetActor
  ...
```

[1] An extensive discussion about encapsulation in games, as well as an optimizer for encapsulated code in Casanova can be found in [?]

```
  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
  ref TSource :TargetActor

  Output0 : Option<T0>
  Output1 : Option<T1>
  ...
  Outputn : Option<Tn>

  Fire : bool

  ...
  rule Resources.Transfers, Output0, ..., Outputn =
    .| threshold_condition ->
      yield None, Some value0, ..., Some valuen, Fire
      yield None, None, ..., None, false

    .| _ ->
      wait restrictions
      yield Some(some_resources), Output0, ..., Outputn
      yield None, Output0, ..., Outputn
```

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 we define the statements to 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 and then its reset. In the following code SourceActor contains, besides the usual fields, also a reference to an object O of type Object and to DestroyedO.

```
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
  yiel 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] David W Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *Case-based reasoning research and development*, pages 5–20. Springer, 2005.

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

[4] 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.

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

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

[7] 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.