

## Tilburg University

### Resource Entity Action

Abbadi, Mohamed; Di Giacomo, Francesco; Orsini, Renzo; Plaat, Aske; Spronck, P.H.M.; Maggiore, Giuseppe

*Published in:*

Proceedings of the 8th International Computer and Games Conference

*Publication date:*

2014

*Document Version*

Peer reviewed version

[Link to publication in Tilburg University Research Portal](#)

*Citation for published version (APA):*

Abbadi, M., Di Giacomo, F., Orsini, R., Plaat, A., Spronck, P. H. M., & Maggiore, G. (2014). Resource Entity Action: A Generalized Design Pattern for RTS games. In *Proceedings of the 8th International Computer and Games Conference* (LNCS; Vol. 8427). Springer Verlag.

#### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

#### Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

# Resource Entity Action: A Generalized Design Pattern for RTS games

Mohamed Abbadi, Francesco Di Giacomo, Renzo Orsini,  
Aske Plaat, Pieter Spronck, and Giuseppe Maggiore  
E-mail: {mabbadi,fdigiacom,orsini}@dais.unive.it, {a.plaat,p.spronck}@uvt.nl,  
maggiore.g@nhtv.nl

Università Ca' Foscari DAIS - Computer Science, Venice, Italy  
Tilburg University, Netherlands  
NHTV University of Applied Sciences, Breda, Netherlands

**Abstract.** In Real-Time Strategy (RTS) games, players develop an army in real time, then attempt to take out one or more opponents. Despite the existence of basic similarities among the many different RTS games, engines of these games are often built ad hoc, and code re-use among different titles is minimal. We created a design pattern called “Resource Entity Action” (REA) that abstracts the basic interactions that entities have with each other in most RTS games. This paper discusses REA and language abstraction and implementation using the Casanova game programming language. Our analysis shows that not only the pattern forms a solid basis for a playable RTS game, but also that it achieves considerable gains in terms of lines of code and runtime efficiency. We conclude that the REA pattern is suitable approach to the implementation of many RTS games.

## 1 Introduction

Real-time strategy (RTS) games have been highly popular for decades. As outlined by the ESA[1], RTS games are registering strong sales and a large number of play hours. Commercial RTS games are written by game developers of different backgrounds: from *large studios* to smaller independent developers of *indie games*. Indie developers [7] typically consist of small teams and their games are known for innovation [8], creativity [10] and artistic experimentation [3]. RTS games are also built as “serious games” [2], used for training and education, and as “research games” [4].

In general, the building of games is an expensive venture [5]. This is particularly conflicting for indie developers and developers of serious and research games, who usually have access to few resources. They would benefit of cost-effective development methodologies for games, through the identification and automation/reuse of common patterns in games. Surprisingly, from a survey of game development research and literature, we noticed a lack of studies of

abstract patterns which characterize games, in particular RTS games. This motivates our research question: *to what extent can we capture the commonalities of RTS games in a re-usable design pattern?*

Section 2 discusses the essential elements of an RTS game. Section 3 specifies the *Resource Entity Action* (REA) design pattern [6] that captures these essential elements. Section 4 describes how the pattern is implemented as a language extension of the Casanova game programming language [9]. The language extension is purely declarative, using semantics that resemble SQL, providing an intuitive adoption for most programmers. We implemented the extension in a full-fledged RTS, which we discuss and analyze in Section 5. In Section 6 we make an analytic comparison between several other available solutions for making RTS games and our own.

## 2 Essential elements of RTS games

RTS are a variation of strategy games where two or more players achieve specific (often conflicting) objectives by performing actions simultaneously in real time. The typical elements which arise from this genre are *units* (characters, armies), *buildings*, *resources* and *battle statistics*. Players command units to perform different types of actions. These actions can affect several entities in the game world.

Units and buildings are the entities that the players control to achieve their objectives. Units usually fight or harvest resources, while buildings may be used to create new units or research upgrades. Resources are gathered from the playing field and fuel the economy of the game entities. Battle statistics determine the attack and defense abilities of units in a fight. This taxonomy of the elements of a RTS game can be applied successfully to multiple games: Starcraft, C&C, and Age of Empires all feature units, buildings, resources, and battle statistics, among other elements.

In order to arrive at our design pattern we will now apply a simplification. Battle statistics can be interpreted as resources, so that “the life of a unit is the cost for killing it, payable in attack power.” We can also merge units and buildings together into a new category called *entity*. This leads us to a simpler view of an RTS as a game that is based on Resources, Entities and Actions:

1. Resources: numerical values in the battle and economic system of the game. In this group we find the *attack*, *defense*, and *life* patterns of entities. Resources also cover building materials and cost of production, deployment of units, development of new weapons, etc. (Resources are scalars.)
2. Entities: container for resources. They have physical properties and, as for the game logic, the difference among them is only the interactions. These interactions take place with resource exchanges through the actions. (Entities are vectors.)
3. Actions: resource flow among entities. Our model can be viewed as a directed weighted graph where the nodes are the entities, the weights are the amounts

of exchanged resources, and the edges are the actions, that is, the elements which connect entities to one another. (Actions are transformation matrices.)

This leads us to a more precise research question: *how do we model Resources, Entities and Actions?*

### 3 The REA design pattern

In this section we will define a model for an algebra to show that the REA (Resource Entity Action) model can be reduced to a problem of linear algebra. We then show how games that use this model can be further simplified by linguistic constructs. In an RTS game we can think of the game world as a collection of entities. Entities perform an action by exchanging resources with one another, thus the resources may be stored in a vector and resource exchange may be expressed by matrix algebra.

#### 3.1 Action algebra

As described in Section 2, we can consider an action as a flow of resources from a source entity to one or more target entities. We require that each entity has a resource vector, which contains the current amount of resources of the entity. The resource vector is sparse since most actions involve only few resource types. An action is expressed by a transformation matrix  $A$  which determines how a resource is passed to another entity for that particular action.

Let us consider a set of target entities  $T = \{t_1, t_2, \dots, t_n\}$  which are the targets of the action and a source entity  $e$ . Each entity  $t_i$  (including the source entity type) owns a resource vector  $\mathbf{r}_i = (r_{i_1}, r_{i_2}, \dots, r_{i_m})$  while the source entity owns also a transformation matrix  $A$  of size  $m \times m$ , which defines how the  $h$ -th component (i.e., resource) of the resource vector is affected by the interaction with the  $k$ -th resource. We also consider an integrator  $dt$  which contains the time difference between the current frame and the previous one. We then compute  $\mathbf{w}_e = (w_{e_1}, w_{e_2}, \dots, w_{e_m}) = \mathbf{r}_s \times A \cdot dt$ . From the definition of matrix multiplication, it immediately follows that each component of  $\mathbf{w}_e$  represents how the  $h$ -th resource will change by applying the effect of all the other resources to it. Now we compute the vector  $\mathbf{r}'_i = \mathbf{r}_i + \mathbf{w}_e \forall e_i \in E$  which will now replace the resource vector in each target entity.

For instance, consider the action of a spaceship entity using laser to damage (resource) an enemy spaceship (entity). This involves a vector resource of two elements: laser and life points. The action must transfer laser points to subtract from the enemy life points. Let us assume the vector resource of the targeting ship is  $r_s = (20, 500)$  and the vector resource of the targeted ship is  $r_t = (15, 1000)$ .

Let the transformation matrix be  $A = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$  which means that the source entity will affect the life of the target with a negative number of laser points. Thus  $w_e = r_s \times A \cdot dt = (20, 500) \times A \cdot dt = (0, -20) \cdot dt$ . At this point, assuming  $dt = 1$  second, we have  $r'_t = r_t + w_e = (20, 1000) + (0, -20) \cdot dt = (20, 980)$ .

### 3.2 A declarative language extension

The REA design pattern is modeled using the action algebra. We will now describe a language extension that implements this design pattern/algebra for the Casanova game programming language [9]. The language extension is purely declarative. Its semantics are described using the SQL query language, which has the advantage of familiarity to most programmers.

Implementing the action algebra may be done using an abstract class which contains an abstract method which performs the action. Each action is a class which extends the previous abstract class and implements the abstract method. This method will fetch the world looking for the information needed to find what entities are affected by the action execution. Each entity of the game will have a collection of actions it can perform, automatically run by Casanova.

We still lack a way to identify the set of target entities  $T$  given a source entity and its action. A solution that lets the programmer implement class inheritance and abstract methods will produce a lot of boilerplate code, where the common behavior is scanning the game world in search of entities satisfying certain properties needed to apply the effect of an action. The reason is that the library lacks knowledge of the game definition, which must be compensated for by the game programmer.

To avoid boilerplate code to a great extent, we can try to hide repeated patterns, by capturing it as a language construct with its syntax and semantic extending Casanova. Such an extension allows us to alleviate the problem of finding entities because, at compile-time, we can explore the shape of the game world and generate the appropriate world exploration code. To do so we add a new type definition: the *action*. An action is a declarative construct which is used to describe not only the resource exchange between entities, but also what kinds of entities participate in the exchange. The resource exchange is based on *transfers* (Add, Subtract, and Set), while the target determination is based on *predicates*: we filter the game world entities depending on their types, attributes and radius (specifying the distance beyond which the action is not applied). Some actions, called threshold actions, are not continuous and make use of special predicates to delay the execution (Output) until certain conditions are met.

Using actions it is possible to specify an exchange of resources in a fully declarative manner, so that the developer does not have to rewrite similar pieces of code ad hoc for each action.

## 4 Action syntax and semantics

We now give the syntax and semantics of actions in Casanova. The grammar allows the definition of actions, which make up the body of spacial Casanova entities which act as placeholders for actions. When an entity contains such an action, the Casanova runtime will apply it to all appropriate targets.

#### 4.1 Action Grammar Definition

In this section we will define formally the grammar definition for actions. To better clarify the use of this grammar, we will first provide a taxonomy for our actions. We divide our actions into three kinds: (1) constant transfer actions, (2) mutable transfer actions, and (3) threshold actions.

**Constant Transfer:** Constant transfer actions update the target fields with a constant value or a value taken from one of the source fields. The source field is not affected by the resource transfer. An example of a constant transfer action is a defense tower shooting an arrow at an infantry unit.

```
TARGET Infantry; RESTRICTION Owner <> Owner; RADIUS 1000.0;
TRANSFER CONSTANT Life - ArrowDamage;
```

**Mutable Transfer:** Mutable transfer actions are used when the resource exchange affects not only the target, but also the source entity. The source field is updated depending on the used operation: if we add a resource to the target, then the same amount is subtracted to the source field, if we subtract a resource then the same amount will be added while if we set the target to a value the source is not changed. An example of a mutable transfer action is a spaceship transferring minerals from its holds to a shipyard.

```
TARGET Shipyard; RESTRICTION Owner = Owner; RADIUS 150.0;
TRANSFER MineralStash + Minerals;
```

**Threshold Action:** Threshold actions follows the same transfer semantics of the previous two types of actions. In addition, they have a collection of threshold values and output operations. The output operations are executed once when all the threshold values are reached. The threshold values are on fields belonging to the source entity. The output operations modify only fields of the source entity following the same semantic of the transfer operations. An example for a threshold action is a worker building a town hall. When the `integrity` of the town hall reaches 100, a flag `completed` is set (which is one of its fields) which warns the system to replace the partial constructed building with the complete building.

```
TARGET ConstructionTownHall; RESTRICTION Owner = Owner;
RADIUS 10.0; TRANSFER CONSTANT Integrity + 1.0; THRESHOLD
Integrity = 100.0; OUTPUT Completed := true
```

Now we give a formal definition for the grammar instances presented in the examples above. In the following formal grammar definition we used reference to Casanova types: `Casanova Entity` is an entity in the game world represented as a record with its fields; the special keyword `Self` is used to point the entity owning the action as one of its fields. To define the grammar we use the extended Backus-Naur form.

```

<Action> ::= TARGET <TARGET LIST> <RESTRICTION LIST> [<RADIUS
    CLAUSE>] <TRANSFER LIST>
    <INSERT LIST> [<THRESHOLD BLOCK>]
<TARGET LIST> ::= <ACTION ELEMENT>+
<ACTION ELEMENT> ::= Casanova Entity | Self
<RESTRICTION LIST> ::= {<RESTRICTION CLAUSE>}
<RESTRICTION CLAUSE> ::= RESTRICTION Boolean Expression of <
    SIMPLE PRED>
<SIMPLE PRED> ::= Self Casanova Entity Field (= | <>) Target
    Casanova Entity Field
<TRANSFER LIST> ::= {<TRANSFER CLAUSE>}
<TRANSFER CLAUSE> ::= (TRANSFER | TRANSFER CONSTANT)
    (Target Casanova Entity Field) <Operator> ((Self Casanova
    Entity Field) | (Field Val)) [* Float Val]
<Operator> ::= + | - | :=
<RADIUS CLAUSE> ::= RADIUS (Float Val)
<INSERT LIST> ::= {<INSERT CLAUSE>}
<INSERT CLAUSE> ::= INSERT (Target Casanova Entity Field) ->
    (Self Casanova Entity Field List)
<THRESHOLD BLOCK> ::= <THRESHOLD CLAUSE>+
<OUTPUT CLAUSE>+
<THRESHOLD CLAUSE> ::= THRESHOLD
    (Self Casanova Entity Field) Field Val
<OUTPUT CLAUSE> ::= OUTPUT
    (Self Casanova Entity Field) <Operator> ((Self Casanova
    Entity Field) | (Field Val)) [* Float Val]

```

## 4.2 Formal semantic definition

Given the fact that our actions resemble queries on entities, we specify their semantics as translation semantics to SQL. This allows us to leverage existing discussions on SQL correctness [12].

In defining our translation rules formally, we consider a set  $T = \{t_1, t_2, \dots, t_n\}$  of target types and a source entity type  $s$ . In all actions we select a subset of targets in each  $t_i$ , on which applying the action, using restriction conditions (if any exist, otherwise the targets of type  $t_i$  are used). After that we apply the resource transfer (which, as explained above, can be either from a constant field or a mutable field).

We assume that each entity type is represented by an SQL relation and that there exists a key attribute called **Id** for each relation. We now consider each of the three translation cases, based on the taxonomy given above, separately. In the following translation rules we will use, for greater clarity, notations inside the SQL code taken from the Backus-Naur form for grammar definitions such as [expr] to denote an optional expression or "|" to denote a choice between expressions. We also extend the SQL grammar with a global variable  $dt$  which is the time difference between the current and the last game frame. In this way

the increment of the entity attribute values are proportional to the elapsed time. All types of actions evaluate the predicates in the restriction conditions and apply a filter to their targets. All targets further than the radius are automatically discarded when executing the action. The transfer predicates are executed immediately on all filtered targets.

**CONSTANT TRANSFER:** In constant transfers we must update each target  $t_i$  satisfying the restriction conditions with the value in the source fields or constant values specified in the transfer clause. For simplicity, we will assume that constant values will be stored as attributes of the source entity.

Let us consider a set of resource attributes  $A = \{a_{j_1}, a_{j_2}, \dots, a_{j_m}\}$  of the source entity used to update the target  $t_i$ . We have to compute the contribution of all sources of the same type on the target  $t_i$ . We want to produce a relation whose tuples represent the target id followed by the total amount of resource  $a_{j_r}$  to transfer, called  $\Sigma_r$ :

Transfer				
$ID$	$\Sigma_1$	$\Sigma_2$	$\dots$	$\Sigma_m$

The following SQL instruction implements the relation definition above:

```

SELECT   $t_i.id$ , SUM( $s.a_{j_1}$ ) AS  $\Sigma_1$ ,
        SUM( $s.a_{j_2}$ ) AS  $\Sigma_2$ , ...,
        SUM( $s.a_{j_m}$ ) AS  $\Sigma_m$ 

FROM    Target  $t_i$ , Source  $s$ 
WHERE   <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY  $t_i.id$ 

```

Now  $\forall t_i \in T$  we must update the target attributes  $A' = \{a_{t_1}, a_{t_2}, \dots, a_{t_m}\}$  using one of the target operators defined in the grammar (Set, Add, Subtract) with the attributes of the previous relation scheme.

```

WITH    Transfer AS (
        SELECT   $t_i.id$ , SUM( $s.a_{j_1}$ ) AS  $\Sigma_1$ ,
                SUM( $s.a_{j_2}$ ) AS  $\Sigma_2$ , ...,
                SUM( $s.a_{j_m}$ ) AS  $\Sigma_m$ )

FROM    Target  $t_i$ , Source  $s$ 
WHERE   [<RESTRICTION LIST>] [AND <RADIUS CLAUSE>]
GROUP BY  $t_i.id$ )
UPDATE  Target  $t_i$ 
SET      $t_i.a_{t_1} = u.\Sigma_1$  |  $t_i.a_{t_1} = t_i.a_{t_1} + u.\Sigma_1 * dt$  |  $t_i.a_{t_1} =$ 
 $t_i.a_{t_1} - u.\Sigma_1 * dt$  \
...
FROM    Transfer  $u$ 
WHERE    $u.id = t_i.id$ 

```



**MUTABLE TRANSFER:** In a mutable transfer the field of the source involved in the resource transfer must be updated depending on the applied transfer operator. If the operator is Add then the resource is subtracted from the source field and added to the target field proportionally to  $dt$ . If the operator is Subtract then the resource is subtracted from the target field and added to the value of the source field proportionally to  $dt$ .

The first step in translating this semantic rule is finding how many targets (if any) are affected by each source entity, in order to obtain the following relation scheme:

TotalTargets	
<i>Source ID</i>	<i>TargetCount</i>

The SQL code implementing the previous scheme is the following:

```
TotalTargets =
SELECT  s.id,COUNT(*) AS TargetCount
FROM    Source s, Target t1, Target t2,...,Target tn
WHERE   <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY s.id
HAVING  COUNT(*) > 0
```

Now  $\forall t_i \in T$  we need to obtain a relation storing what target each of the source entity is affecting and the count of affected targets. The following relation scheme describes what we have just informally explained:

OutputSharing		
<i>Source ID</i>	<i>Target ID</i>	<i>Output Sharing</i>

For brevity in the code below it has been used the notation RelationName = [...] which means the code for that table has been previously defined. The following SQL code implements the previous scheme:

```
OutputSharing =
SELECT  *
FROM    TotalTargets c, SourceOutput c1
WHERE   c.s_id = c1.s_id
        AND SourceOutput =
            SELECT  s.id AS s_id,t_i.id AS t_id
            FROM    Source s, Target t_i
            WHERE   <RESTRICTION LIST> [AND <RADIUS
                CLAUSE>]
        AND TotalTargets = [...]
```

Each target attribute receives an amount of resources equal to the total transferred resources divided by the number of targets receiving that resource from the source. The latter value can be read in the table obtained with the SQL code above. Formally, let  $a_{jk} \in A$  be the resource attribute containing the total amount to be transferred and  $c$  the number of targets affected by the resource

transfer, then each target receives  $R_k = a_{j_k}/c$  resources from each source. The values needed to compute  $R_k$  can be obtained both from *OutputSharing* and the Source relation. The complete SQL code to update the target  $t_i$  is the following:

```

WITH      Transfer AS (
          SELECT  ti.id, SUM(s.aj1 / o.TargetCount) AS Σ0, SUM(s.aj2
                / o.TargetCount) AS Σ2, ..., SUM(s.ajm / o.
                TargetCount) AS Σm
          FROM    Source s, Target ti, OutputSharing o
          WHERE   OutputSharing = [...] AND s.id = o.s_id AND t
                .id = o.t_id)
          GROUP BY ti.id
UPDATE    Target ti
SET       ti.at1 = u.Σ1 | ti.at1 = ti.at1 + u.Σ1 * dt | ti.at1 = ti.at1 - u.
          Σ1 * dt
...
FROM      Transfer u
WHERE     ti.id = u.id

```

To update the Source relation we use a relation similar to the one use to update the target, but this time there is no need to save the count of the affected targets but just to check if the source has at least one target, otherwise it is not updated.

```

WITH      TotalTransfer AS (
          SELECT  s.id, s.aj1, s.aj2, ..., s.ajm
          FROM    Source s, Target t1, ..., Target tn
          WHERE   <RESTRICTION LIST>
                [AND <RADIUS CLAUSE>]
          GROUP BY      s.id, s.aj1, s.aj2, ..., s.ajm
          HAVING    COUNT(*) > 0)
UPDATE    Source s
SET       s.aj1 = s.aj1 - s.aj1 * dt | s.aj1 = s.aj1 + s.aj1 * dt
...
FROM      TotalTransfer u
WHERE     s.id = u.id

```

**THRESHOLD TRANSFER:** A threshold action is an action defined as the previous two types, i.e. it has a resource transfer definition which is always executed, and a set of threshold conditions that, if met, activate the Output operations, which are always towards the source entity. The attributes of the source entity affected by Output operations can be updated with constant values or values from other attributes in the source entity. In the latter case the transfer is treated as in the mutable transfer case.

Let us consider a set of updating attributes  $U = \{a_{k_1}, a_{k_2}, \dots, a_{k_l}\}$  and a set of attributes to be updated  $U' = \{a_{s_1}, a_{s_2}, \dots, a_{s_l}\}$  in the output operation. We

must first check that all the conditions in the threshold clauses are met, then we have to update the attributes in the source entity appropriately.

```

WITH      TotalOutput AS(
SELECT    s.id, s.ak1, s.ak2, . . . , s.akl
FROM      Source s
WHERE     <THRESHOLD CLAUSE 1>
          [AND <THRESHOLD CLAUSE 2>]
          .
          .
          .
          [AND <THRESHOLD CLAUSE l>])
UPDATE    Source s
SET       s.as1 = o.ak1 | s.as1 = (s.as1 + o.ak1) * dt; o.ak1 = o.ak1 -
          o.ak1 * dt | s.as1 = (s.as1 - o.ak1) * dt; o.ak1 = o.ak1 + o.ak1 * dt
. . .
FROM      TotalOutput o
WHERE     s.id = o.id

```

### 4.3 Casanova implementation

The process of evaluating actions was added to Casanova, which, using a compiler, generates assembly code specific for each action. The generated code executes the actions at each game frame. Besides, the compiler checks that the targets are valid and that the fields used in all the predicates are contained in those entities. To improve performance an index is built at compile time, to speed up resolution of radius restrictions. The implementation uses type attributes for actions, so the syntax is different even though there is a mapping between elements of the syntax presented here and those of the concrete syntax.

## 5 Case study

We now present an RTS game we used as a case study and the benchmarks that test the action implementation. We call this game “CS” for “case study.” In the game players must conquer a star system made up of various planets. Each planet builds fleets which are used to fight the fleets of the other players and to conquer more planets. A planet is conquered when a fleet of a player is near it and no other enemy fleet is defending it.

### 5.1 Case study

Three actions are required in this game: The first action, called **Fight Action**, defines how a fleet fights enemy fleets in range. The fight action subtracts  $0.5 \cdot dt$  life points to the enemy fleet during every action tick (every frame) which are in range.

```
Fleet = {Position: Rule Vector2;FightAction: FightAction;
        Owner: Ref Player;Life: Var float32;Fight: FightAction }
```

The fight action is defined as follows:

```
FightAction = TARGET Fleet; RESTRICTION Owner <> Owner;
            RADIUS 150.0; TRANSFER CONSTANT Life - 0.5;
```

The action target is an entity whose type is **Fleet**, the condition to execute the action is that the fleet must be an enemy, so the fleet owner must be different as specified in the **Restriction** clause. The **attack range** is 150 units of distance, so the **Radius** will be 150. When attacked 0.5 life points are subtracted at every attack.

The second action is called **BuildAction** and allows a planet to create a ship. In order to build a ship, a planet must gather 10 mineral units. Each planet has a field called **GatherSpeed** which determines how fast it gathers minerals. Every tick the planet mineral stash is increased by this amount. This action is a threshold action where the threshold value is the minerals of the planet. As soon as the threshold value is reached, we set the field **NewFleet** to **TRUE** (it is used by the engine to create a new fleet) and **Minerals** to 0 to reset the counter. The planet and its actions are:

```
Planet = {Position: Vector2;Owner: Rule Ref Player;NewFleet:
        Rule bool;BuildAction:BuildAction;
        EnemyOrbitingFleetsAction : EnemyOrbitingFleetsAction;
        GatherSpeed: float32;Minerals: Var float32 }
```

```
BuildAction =
TARGET Self; TRANSFER CONSTANT Minerals + GatherSpeed;
        THRESHOLD Minerals 10.0; OUTPUT NewFleet := true; OUTPUT
        Minerals := 0.0
```

A Casanova rule is appointed to read the value of **NewFleet** and, if it is true, it spawns a new fleet.

The third action is required to check if a planet can be conquered by a fleet. A fleet can conquer a planet if there is no enemy fleet near it and if it is close enough. Thus the action definition will be the following:

```
EnemyOrbitingFleetsAction =
TARGET Fleet; RESTRICTION Owner Not Eq Owner; RADIUS 25.0;
        INSERT Owner -> EnemyOrbitingFleets
```

The action will add an enemy fleet close enough in a data structure used by a Casanova rule to change the owner of the planet.

Even the concept of drawing lasers can be implemented using the **INSERT** clause simply adding it to **FightAction** which inserts in a list all the targeted ships positions. In this way we can draw a laser from the source position to the target position. We omit this aspect for brevity.

## 5.2 Evaluation

We evaluate the performance of actions with the CS case study of the previous subsection and with an additional example. The additional example is a shooter where the player moves a ship and fires at incoming asteroids. We used actions to model the damage interactions between projectiles and asteroids. Table 1 shows a code length comparison between the implementation with actions and standard Casanova rules for CS, Asteroid Shooter and an expanded version of CS with more complex rules, which is used only for comparing the code length and not the performances since the first two samples are enough.

We note that in games with basic dynamics the code saving is not very high due to the fact that the repeated patterns are not many. The advantage of using actions becomes evident in a game with actions involving many types of targets, such as the expanded CS game: without the use of actions, the code related to each interaction would be repeated for each type of target while, with actions, the code is always the same. Furthermore we managed to drastically increase the performance of the game logic: as we can see from Figure 1, using R.E.A. (labeled “with actions”) gives us a speedup between 6 times and 25 times thanks to automated optimizations in the query evaluation. We also note that our implementation is flexible and general since it is possible to use our actions to express a behavior, such as a projectile collision, which is not strongly related to RTS games, since in those games player do not shoot manually a target but it is the game entity which automatically attacks nearby targets.

**Table 1:** *CS (case study), Asteroid Shooter and Extended CS code length*

	Game Entities	Rules	Actions	Total
<i>CS with REA</i>	41	71	19	131
<i>CS without REA</i>	40	90	0	130
<i>Asteroid shooter with REA</i>	33	33	6	72
<i>Asteroid Shooter without REA</i>	34	44	0	78
<i>Extended CS with REA</i>	135	138	40	313
<i>Extended CS without REA</i>	135	328	0	463

## 6 Comparison with related systems

Currently there are many commercial and open source solutions for developing RTS games which result to be often too specific, thus inflexible or not scalable. When users would like to extend these frameworks, this often turns out to be difficult, if not impossible, unless they change the entire structure of the project by changing the structures of entities and the connections among them. There are not many specific RTS engines but some of the most common used are listed below.

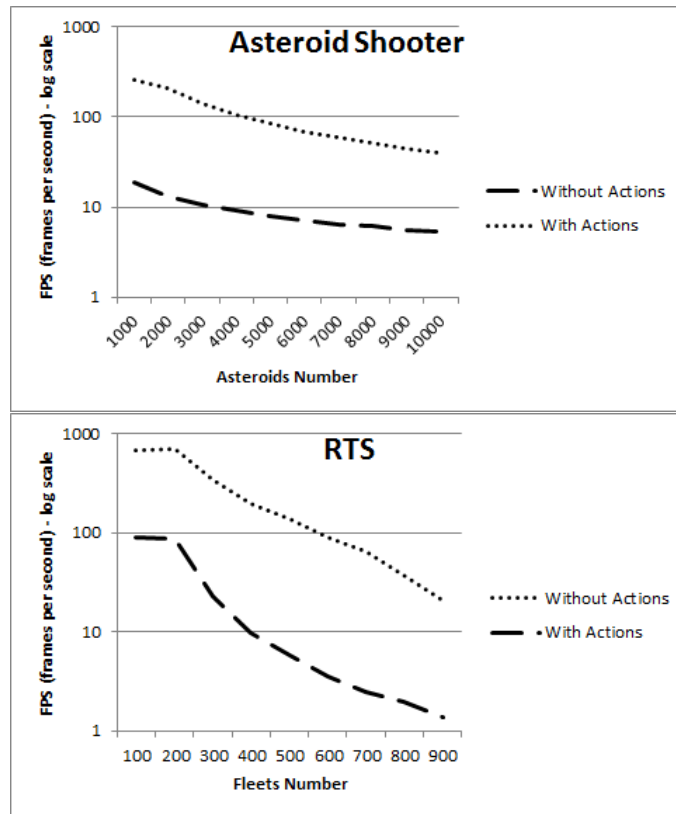


Fig. 1: Frame rate as a function of numbers of entities

**Game maker:** Game maker is a tool which joins the visual development with a limited scripting language. The scripting language allows only the use of strings and real numbers, possibly indexed as arrays. However, it is neither possible to pass an array as a script argument nor accessing it with a pointer except by passing a string holding the name of the array itself. R.E.A. instead is an extension of a well defined and structured programming language like Casanova with no such limitations and workarounds.

**ORTS – Open real-time strategy engine:** ORTS is a domain specific language for making RTS games based on scripts. The language of the scripts is limited; what is not supported by its primitives must be written in C. However, this lack of expressiveness is compensated by being domain specific. ORTS is not designed to be very general because it is specific only for the RTS genre, and in particular for RTS's without an articulated logic. Finally, native optimization, which is provided by our solution, is not possible in ORTS, as explained above, unless the developer codes it by himself.

**Spring engine:** Spring engine is a framework for creating RTS games. The engine specifies predefined boundaries on game dynamics, which cannot be extended. The developer has to learn a long series of keywords. Moreover, getting out of the predefined context, requires to code in a different semantic level using scripting languages such as LUA. However, Spring engine is a good RTS framework which implements a wide variety of options and, in some cases, native optimization (such as spatial optimization for collision detection). Spring engine also presents the same problems, as for the scripting language, of the other engines listed above.

## 7 Conclusions

In this paper we are concerned with the problem of finding a general way to define RTS games. This problem manifests itself in the boilerplate code that developers have to write when they are rewriting common patterns identified during the generalization process.

The results are:

- A **design pattern** for making RTS games where the interaction among entities can be reduced to a dynamic exchange of resources.
- An expressive, high performance **language extension** to Casanova with a compiler and an appropriate grammar with new syntax and semantic. The language extension is purely declarative. Its semantics resemble SQL.
- An evaluation with three examples provides evidence for increases in programming efficiency.
- The evaluation shows an increase in run time efficiency of 6 to 25 times for the Casanova language, using a native code compiler/optimizer.

Even better results could be obtained with an actual access plan optimizer that would increase the performance when exploring the structure of both the action query and the entity structure. Given the significant results on position indexing, the chance of defining multi-attribute indexes would increase the performance. Finally, a system like F# quotations [11] might be used to increase the expressiveness of the actions.



## Bibliography

- [1] Essential Facts About the Computer and Video Game Industry 2011, 2011.
- [2] Clark Aldrich. *The Complete Guide to Simulations and Serious Games: How the Most Valuable Content Will Be Created in the Age Beyond Gutenberg to Google*. Pfeiffer & Co, 2009.
- [3] Erik Andersen, Yun-En Liu, Rich Snider, Roy Szeto, and Zoran Popović. Placing a value on aesthetics in online casual games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11*, pages 1275–1278, New York, NY, USA, 2011. ACM.
- [4] Michael Buro. Real-time strategy games: a new ai research challenge. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1534–1535. Morgan Kaufmann Publishers Inc., 2003.
- [5] Michael Buro and Timothy Furtak. On the development of a free rts game engine. In *GameOn'NA Conference*, pages 23–27. Montreal, 2005.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [7] Juan Gril. The state of indie gaming, 4 2008.
- [8] Johan Kristiansson. Interview starbreeze studios johan kristiansson. <http://www.develop-online.net/features/478/Interview-Starbreeze-Studios-Johan-Kristiansson>, 5 2009.
- [9] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems, EICS '12*, pages 287–292, New York, NY, USA, 2012. ACM.
- [10] David Michael. *Indie Game Development Survival Guide (Charles River Media Game Development)*. Charles River Media, 2003.
- [11] Don Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.
- [12] Günter von Bültzingsloewen. Translating and optimizing sql queries having aggregates. *Proc. 13th Int. Con. VLDB*, pages 235–243, 1987.