Automated networking in Meta-Casanova

Francesco Di Giacomo[1], Mohamed Abbadi[1], Agostino Cortesi[1], Pieter Spronck[2], and Giuseppe Maggiore[3]

[1] Universita' Ca' Foscari, Venezia,
`francesco.digiacomo@unive.it, cortesi@unive.it, mohamed.abbadi@unive.it`
[2] Tiburg University, `p.spronck@uvt.nl`
[3] Hogeschool Rotterdam, `maggg@hr.nl`

Abstract. Multiplayer support is a desirable feature in commercial games since it allows to extend the longevity of a title. The same importance can be found in several examples of serious games, as they are often used in training where several individuals must cooperate to reach a common goal. Having smaller resources than big entertainment companies, serious games developers cannot afford to develop ad-hoc networking code and need to rely on game engines. On the other hand, game engines require deep knowledge and do not overcome all the problems of hand-made networking implementations. For this reason, it is desirable to express networking behaviours in a dedicated language, or Domain-specific language, that provides the necessary abstractions, but this approach suffers from having to implement a compiler for the language, which is hard to extend. In this paper we provide an alternative approach based on meta-compilation defined in Metacasanova, a meta-compiler where it is possible to specify the formal semantics of the language. We show that using Metacasanova notably reduces the effort of developing a Domain-specific language while keeping the same advantage in terms of quality of code.

1 Introduction

Adding multiplayer support to games is a desirable feature to increase the game popularity by attracting millions of players [7]. The benefit for the popularity is given by the fact that, after completing the singleplayer experience, the user may keep playing the game online in cooperation or against other players, in different modes that are more interesting and challenging than their singleplayer counterparts [13]. Multiplayer games are very sensitive to network delay (latency), packet loss, and bandwidth limits, thus requiring heavy optimizations that are different and depend on the specific instance of the game. For instance, a Turn-based strategy game will require less optimization with respect to bandwidth usage or delays, because the data transmission happens once, while in a First-person shooter, which gameplay is based on fast-paced action, low latency is paramount. In addition to games developed for the entertainment sector, networking is a crucial feature also in the field of serious games. However, serious games developers do not have access to the same

amount of resources and manpower available at big companies such as EA or Ubisoft, while having to deal with the same degree of complexity in their products.
Put some examples of existing serious games
For this reason, manually implementing the networking module is not a feasible solution in the field of serious games, as serious games developers have to rely on game engines, such as Unity 3D or Unreal Engine, which offer a built-in API to deal with networking. Although being a better approach than developing a hand-made networking protocol for a specific game, game engines require extensive knowledge to be used effectively and require, to some extent, to still deal directly with low-level issues related to data transmission and remote procedure calls. This problem can be solved in part by developing a Domain-specific language including abstractions to define the behaviour of the game in a networking scenario. Choosing this approach requires to implement a compiler for the language, which can be argued to be as time-consuming as developing a hand-made networking implementation. In this paper we propose an alternative approach to the development of a hard-coded compiler for a Domain-specific based on meta-compilation, which has been proven to be an effective technique in different works [11, 6], in order to have the benefits of a domain-specific language for networking and, at the same time, a lower development effort.
Complete with summary of the paper

2 Challenges of multiplayer games

In this section we explore several existing solutions to designing a networking layer for multiplayer games. For each solution we list the main advantages and drawbacks measured in terms of four characteristics:

- *Loose coupling*: the measure of how closely two modules are.
- *High cohesion*: the degree to which the elements of a module belongs together.
- *Code re-usability and portability*: how it is possible to re-use the code for a different similar game and how much of the code can be ported to this version.
- *Extensibility*: the ease of extending the networking implementation with additional functionality.

Finally, we present our alternative solution and formulate the problem statement.

2.1 Preliminaries

Architecture of networking in games, local view of the "real" game state, centralized server vs p2p.

2.2 Manual implementation

Manual implementation of a networking layer for multiplayer games is the most common choice for big development video game studios. Notable examples can be found in games such as Starcraft, Age of Empire, Supreme Commander, and Half-Life. This solution is tightly bound to the behaviour of game entities and requires to implement all the networking components by hand, from the data transmission primitives to the synchronization logic and client-side prediction. In this scenario the data transmission and synchronization will be dependent on the structure and behaviour of the entities. Indeed, the transmission must take into account the nature of the data describing the game entity (value or reference) and the action it might perform during the game. For instance, let us assume that we want to synchronize the action of shooting with a gun, the game creates a projectile locally when the player presses the button to shoot and that information must be shared with all the players. Furthermore, the modules functionality is not dedicated to a specific task but interconnected with that of other game components. Thus, this solution is affected by two main problems: *tight coupling*, *low cohesion*. Notable examples of these problems were encountered during the development of Starcraft [15], where it was reported that the game code for specific units (including multiplayer behaviours) were branched and completely diverged from the rest of the program, and Half-Life [4], where the transmitted data was bound to a specific pre-defined data structure and any change to the game had to be reflected there. Moreover, the networking code flexibility and the chance of re-use it in different games is non-existent as it is strictly bound to the specific implementation of the game.

2.3 Game Engines

Game engines overcome the problems of *low cohesion* by providing built-in modules and functionalities to manage the network transmissions for multiplayer games. Unity Engine [8] and Unreal Engine [9] are both commercial game engines that support networking. Their approach is centred around a centralized server approach, where the "real" state of the game is stored on the server and the clients only see its local approximation, which must be periodically validated and possibly corrected. The game engine offers the chance of defining what entities should be replicated on the clients and the synchronization is automatically granted by the engine itself, with the possibility of customizing some parameters, such as the update frequency and the prediction methods. Further customization can be achieved by using *Remote procedure calls* (RPC's) by specifying (*i*) that a specific function is called on the server and executed by the clients (for instance, to control events that do not affect gameplay, such as graphical effects), by(*ii*) that a function is called on the client and executed on the server (for instance, to perform actions), and (*iii*) multi-cast calls (executed on all the clients and the server).
The approach based on game engines grants *high cohesion*, because it allows to separate the logic of the game from the logic of the networking layer by providing library-level abstraction. The problem of *tight coupling* persists because the game logic will be still connected to the several networking components of the game engine. This solution also offers little to no code re-usability and portability, since anything related to the networking part (and also in general to the whole game implementation)

will be bound to the specific game engine, thus, for instance, migrating from Unity Engine to Unreal Engine would require to rewrite most of the code.

2.4 Language-level primitives

Another option is to implement primitives at language level, that is to build a *domain-specific language* to support networking primitives defining network synchronization behaviours and data transmission. This approach grants *high cohesion* and *low coupling*, since the details of the networking are hidden and abstracted away by the language itself and the game will only interface with the language layer. This approach was explored in scientific works such as the *Network Scripting Language* (NSL) [14] or Casanova Language [10]. This approach grants also code re-usability and portability, as the networking code is bound only to the language and not to a particular engine or specific implementation of the game. For this reason, code related to components of the game that exhibits similar behaviours can be re-used for other games. This approach requires to build a compiler (or interpreter) for the language, which makes extending the networking layer with additional functionalities difficult as any change to the networking language must be reflected in the implementation of the compiler itself.

2.5 Meta-compilation

In the previous considerations we analysed different techniques to implement network communication for a multiplayer game. In order to evaluate the goodness of each solution, we evaluated it in terms of loose coupling, high cohesion, code re-usability and portability, and extensibility. Table 1 summarizes this evaluation, pointing out whether a specific technique grants one of the properties. The solution that grants the highest number of properties is the one using abstractions defined in a Domain-specific language. As stated above, this solution still lacks extensibility as all further extensions to the language must be reflected directly in the compiler implementation. Thus, the capability of extending the language is strictly connected with the capability of extending its compiler with additional language constructs. This is known to be a hard and time-consuming task, as a compiler is made of several inter-operating components responsible for the translation steps and the semantics analysis of the program [3, 11]. Regardless of this, the only creative aspect of defining a compiler is not the implementation of such modules, that always follows the same logic, but only the definition of the language constructs [5]. Meta-compilers are programs that are able to take as input the definition of a language, defined in a meta-language, a program written in that language, and generate executable code. Using a meta-compiler to define the semantics of a Domain-specific language has been proven useful in several situations, for example in [11, 6], but to our knowledge never attempted in the scope of creating language-level primitives for multiplayer games development. Given these considerations we formulate the following problem statement:

Problem statement: *Given the formal definition of language-level networking primitives, to what extent using a meta-compiler eases the process of integrating them into an existing Domain-specific language?*

In what follows, we will present Casanova 2, an existing Domain-specific language, and Metacasanova, a meta-compiler used to give a re-implementation of Casanova 2. We then proceed to present a networking architecture defined in Casanova 2 and we give the semantics of its primitives. We finally show an implementation of such semantics in Metacasanova and we compare the code length with its respective implementation counterpart in the Casanova 2 hard-coded compiler.

Technique	**Loose coupling**	**High Cohesion**	**Re-usability/portability**	**Extensibility**
Manual implementaion	✗	✗	✗	✗
Game engine	✗	✓	✗	✗
Language-level primitives	✓	✓	✓	✗

Table 1. Advantages and disadvantages of networking implementation techniques

3 Networking architecture and semantics

In this section we give an overview of Casanova 2, a declarative domain-specific language oriented towards game development. In this section we just give a brief informal overview of the language; for further technical details we point the reader to [1, 2]. We then present the language extension to include networking abstractions, which prototype was presented in [10] and hard-coded in its compiler, and we finally present its formal semantics.

3.1 Overview of Casanova 2

Casanova 2 is a declarative language oriented to game development. A program in Casanova 2 consists of a set of entities that define the dynamic elements of the game. These range from elements which interact or are interacted with by the player, such as characters, weapons, or interactive scene elements, to elements not directly operable such as bullets or non-playable characters. The entities are organized into a tree structure, at which root you find a special entity called *World*. Each entity defines a set of fields, in the fashion of a class, and a set of rules that define its temporal evolution. Unlike object-oriented programming, entity fields are not directly modifiable through variable assignments (although they can always be read), but only through rules: each rule takes as input a domain, which is a subset of fields that it can modify, and they can be updated only by calling a dedicated statement called `yield`. If a rule tries to update a field outside its domain, this is captured at type-system level. Moreover, each rule works in the fashion of threads (although the implementation in the back-end of the compiler is completely different,

as it makes use of state machines rather than a scheduler), meaning that they can be interrupted by using built-in abstractions in the language, such as `wait` statements, or the `yield` itself, which stops the rule execution by one game frame. Finally, each rule takes as input a reference to the current entity (`this`) and a parameter `dt` which contains the elapsed time between the current game frame and the previous update.
In the next section we present a language extension for Casanova 2 in order to include abstractions to express networking synchronization.

3.2 Networking architecture

The networking architecture that we present is peer-to-peer based. This means that we do not have a centralized authoritative server that constantly validates the local versions of the game state on the clients, but rather each client is tasked with maintaining and validating a portion of the game state.
In this scenario, we identified the following abstractions that must be provided by the language:

– *Sending and receiving data:* the developer should be able to specify how and what data to send and receive to and from other clients.
– *Connection:* the developer should be able to specify what happens when a new player connects to the game and how the existing players react to this event.
– *Managing the local and remote game state:* the developer should be able to define how local and remote entities evolve on each client.

Below we present the extensions implementing such abstractions.

3.3 Data transmission

The language extension should include primitives to send data. These primitives should allow the programmer to choose whether to send data in a reliable (meaning that the receiver will be sure to receive the data at some point) or an unreliable way (meaning that the data could be lost because of packets losses). To this end, we define the primitives (`send` and `send_reliable`. The programmer should be provided with the means of receiving data, which are given by the primitives `receive` and `receive_many`: the former receive a single entity or atomic value, the second is able to receive a list of entities or values. The `receive` operations can be used in combination with the `yield` statement to simultaneously receive the data and update an entity field. Furthermore, we will be needing of a language primitive that waits until a received value is available and then binds it to a variable, called `let!` to distinguish it from a normal `let` binding. The following snippet illustrates an example using some of these primitives (the `@` operator denotes lists concatenation):

```
world Shooter = {
...
rule master Ships =
  let! receivedShips = receive_many
  yield Ships @ receivedShips
}
```

Handling connections When a new client connects to the game, it instantiates a local copy of the game state. This game state is incomplete, meaning that it does not contain an information about the game state views of the other clients, nor the other clients know anything about the local view of the new client. At this purpose, we extend Casanova 2 rules with two clauses: `connecting` and `connected`. A rule defined as `connecting` is executed only once when a new client connects to the game. On the other hand, a rule marked as `connected` is executed by all clients when a new client connects. An example of this is shown below: a client connects to the game and instantiates a ship locally, which is sent to the other clients. At the same time, the rule defined as `connected` is executed by the existing clients which sends their ships. The data transmission primitives will be discussed further below.

```
world Shooter = {
  ...
  rule connecting Ships =
    yield send_reliable Ships

  rule connected Ships =
    yield send_reliable Ships
}
```

Note that, even if the two rules seem to be identical, the semantics is completely different: the first is run by a connecting client and sends the only local ship instantiated when connecting to the game to all the existing clients, the second is executed by all the other clients, each one sending a list with the existing Ships.

3.4 Local and remote entities

To handle local and remote portions of the game state we must divide accordingly the entities in two sets: those instantiated by the current client, which are under its direct control, and those instantiated by other clients and under their control. To further illustrate this situation, consider a simple shooter game where every client controls a ship that can shoot the other clients' ships. The client can use input devices (such as mouse and keyboard or a gamepad) to perform an action with its ship, but it should be unable to control the other clients' ships. At this purpose the client should send the updates performed locally on the ship it controls, and receive the updates performed at the same time on the other clients for all the other ships. Entities instantiated locally by a client are defined as `master` entities, while entities that were instantiated on remote clients and which copy is replicated in the current client are defined as `slave`. We extend the rules with this two additional identifiers `master` and `slave`. Rules defined as `master` are run only for instances of entities that were locally instantiated by a client, while rules defined as `slave` are run only for instances of entities that were instantiated remotely. Note that rules that are not marked either as `master` or `slave` are always run independently on the entity being instantiated locally or remotely. This could be helpful to perform dynamics that do not need a synchronization with other clients, such as reacting to an interaction with a GUI element. An example of rules defining the dynamics for local entities is shown below:

```
entity Ship = {
   ...
   rule master Position =
     wait world.Input.KeyDown(Keys.W)
     let vp = new Vector2(Math.Cos(Rotation),Math.Sin(Rotation)) * 300.0f
     let p = Position + vp * dt
     yield send p
}
```

3.5 Formal semantics

In what follows we present the operational semantics of the networking language extension that was introduced above. In order to keep consistency with how Casanova semantics was defined [2], we use a rewrite-based definition [12], where the world is transformed into a new one containing an updated version of its fields and rules.
The semantics of Casanova 2 must be extended to reflect the fact that, in a networking scenario, there is a *Network State* (see Section 2.1) consisting of local approximated views of the global state. At this purpose we define this state as a set $N_s = \{S_{c_i} | c_i \in C\}$ where C is the set of clients connected to the game and S_{c_i} is the local game state for client c_i. Each local state is a pair $S_{c_i} = (E_L, E_R)$ of local (instantiated on the client c_i) and remote (instantiated on a different client c_j, $j \neq i$) entities. Each entity $e \mid e \in E_L \vee e \in E_R$ is a set of fields and rules:

```
E = {$Field_1 = f_1, ..., Field_n = f_n$
     $Rule_1 = r_1, ..., Rule_m = r_m$}
```

In order to include the definition of `master`, `slave`, `connecting`, and `connected` rules, we consider their respective sets R_m, R_s, R_c, and R'_c.

Entity update in a networking scenario The update of the game state in Casanova 2 is realised by using a recursive function `tick` that recursively updates all the fields of an entity and, at the same time, executes all its rules through another function `step`.

```
tick(e:E, dt) =
{ Field$_1$=tick(f$_1^m$, dt); ...; Field$_n$=tick(f$_n^m$, dt);
  Rule$_1$=r$'_1$; ...; Rule$_m$=r$'_m$ }
  where
   f$_1^m$, ..., f$_n^m$, r$'_m$ = step(f$_1^{m-1}$, ..., f$_n^{m-1}$, r$_m$)
   .
   .
   f$_1^1$, ..., f$_n^1$, r$'_1$ = step(f$_1$, ..., f$_n$, r$_1$)}
```

In what follows we will refer to `tick` and `step` as defined above, where the `tick` function is extended with an additional argument `c` denoting that it is being performed by client `c`. For further details see [2].

Connecting rule semantics As explained above, a connecting rule is executed once on a client connecting to the game for the first time. Thus, if the client does not yet exist among the clients participating to the game session the rule is executed

```
tick(e:E, dt, c) =
{ Field$_1$=tick(f$_1^m$, dt); ...; Field$_n$=tick(f$_n^m$, dt);
  Rule$_1$=r$'_1$; ...; Rule$_j$ = r$'_j$; ...; Rule$_m$=r$'_m$ }
  where
```

```
    $r_j \in R_c$, c $\notin C$
    $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
    .
    .
    $f_1^j$, ..., $f_n^j$, $r'_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
    .
    .
    $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

On the other hand, if the client already exists among the current clients, the rule has no effect on the entity.

```
tick(e:E, dt, c) =
{ $Field_1$=tick($f_1^m$, dt); ...; $Field_n$=tick($f_n^m$, dt);
  $Rule_1$=$r'_1$; ...; $Rule_j$ = $r_j$; ...; $Rule_m$=$r'_m$ }
  where
    $r_j \in R_c$, c $\in C$
    $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
    .
    .
    $f_1^{j-1}$, ..., $f_n^{j-1}$,$r_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
    .
    .
    $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

Connected rule semantics A connected rule is executed when a new client connects to the game, so when there exists a client that does not belong to the set of connected clients.

```
tick(e:E, dt, c) =
{ $Field_1$=tick($f_1^m$, dt); ...; $Field_n$=tick($f_n^m$, dt);
  $Rule_1$=$r'_1$; ...; $Rule_j$ = $r'_j$; ...; $Rule_m$=$r'_m$ }
  where
    $r_j \in R'_c$, $\exists c' | c' \notin C$
    $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
    .
    .
    $f_1^j$, ..., $f_n^j$,$r'_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
    .
    .
    $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

Again, if no new client has connected to the game, then the rule has simply no effect.

```
tick(e:E, dt, c) =
{ $Field_1$=tick($f_1^m$, dt); ...; $Field_n$=tick($f_n^m$, dt);
  $Rule_1$=$r'_1$; ...; $Rule_j$ = $r_j$; ...; $Rule_m$=$r'_m$ }
  where
    $r_j \in R_c$, $\forall c' | c' \in C$
   $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
   .
   .
   $f_1^{j-1}$, ..., $f_n^{j-1}$,$r_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
   .
   .
   $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

Master rule semantics Master rules are executed on instances of entities that were instantiated on the current client, that is, when the entity belongs to the set of local entities of the current client state.

```
tick(e:E, dt, c) =
{ $Field_1$=tick($f_1^m$, dt); ...; $Field_n$=tick($f_n^m$, dt);
  $Rule_1$=$r'_1$; ...; $Rule_j$ = $r'_j$; ...; $Rule_m$=$r'_m$ }
  where
    $r_j \in R_m$, e $\in E_L$
    $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
    .
  .
  $f_1^j$, ..., $f_n^j$,$r'_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
  .
  .
  $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

Slave rule semantics Slave rules are executed on instances of entities that were instantiated on a remote client, that is, when the entity belongs to the set of remote entities of the current client state.

```
tick(e:E, dt, c) =
{ Field$_1$=tick($f_1^m$, dt); ...; Field$_n$=tick($f_n^m$, dt);
  Rule$_1$=$r'_1$; ...; Rule$_j$ = $r'_j$; ...; Rule$_m$=$r'_m$ }
  where
    $r_j \in R_S$, e $\in E_R$
    $f_1^m$, ..., $f_n^m$, $r'_m$ = step($f_1^{m-1}$, ..., $f_n^{m-1}$, $r_m$)
    .
    .
    $f_1^j$, ..., $f_n^j$,$r'_j$ = step($f_1^{j-1}$, ..., $f_n^{j-1}$, $r_j$)
    .
    .
    $f_1^1$, ..., $f_n^1$, $r'_1$ = step($f_1$, ..., $f_n$, $r_1$)
```

Semantics of data transmission The networking transmission semantics must include the possibility that some messages might be lost during the transmission. At this purpose we introduce a parameter $p_r \in]0, 1[$ for the *network reliability* which represents the probability that a message is lost before reaching its destination. We also use a parameter $p_m \in]0, 1[$ which is a random number generated for each message to simulate the message losses with respect to p_r. In what follows we change the definition of the `step` function to include the fact that there are multiple clients running the rules in the network state, and that a receive on a client usually corresponds to a send on another client.

Sending and receiving unreliable messages The language provides two primitives to send messages, one which reliably sends the message, meaning that it ensures that the receiver receives the message, and one unreliable primitive, with which the message could be lost with no chance of recovering it. In the case of an unreliable send, the client send the message and if received, that is if $p_m \geq p_r$ then the fields of the receiving client are updated with the message value, otherwise the receiver does not receive the message and simply skips to the next statement.

```
step($f_1^{c_i}$},...,$f_n^{c_i}$,{send x; $k^{c_i}$}),$f_1^{c_j}$},...,$f_n^{c_j}$,{yield receive x; $k_{c_j}$}) =
(step($f_1^{c_i}$},...,$f_n^{c_i}$,{$k_{c_i}$}),(x, {$k_{c_j}$})
when $p_m \geq p_r$
```

```
step($f_1^{c_i}$},...,$f_n^{c_i}$,{send x; $k^{c_i}$}),$f_1^{c_j}$},...,$f_n^{c_j}$,{yield receive x; $k_{c_j}$}) = (step($f_1^{c_i}$},...,f
      $_n^{c_i}$,{$k_{c_i}$}),($f_1^{c_j}$,...,$f_n^{c_j}$,{ $k_{c_j}$})
when $p_m < p_r$
```

4 Implementation in Metacasanova

– Introduction to Metacasanova with explanation on rule evaluation.

- Overview of the implementation of Casanova 2 with Metacasanova.
- Implementation of networking in Metacasanova.

Implementation in Metacasanova.

5 Evaluation

6 Conclusion

References

1. Mohamed Abbadi. *Casanova 2, A domain specific language for general game development.* PhD thesis, Universita' Ca' Foscari, 2017.
2. Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi, Pieter Spronck, Giulia Costantini, and Giuseppe Maggiore. *Casanova: A Simple, High-Performance Language for Game Development*, pages 123–134. Springer International Publishing, Cham, 2015.
3. Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques.* Addison wesley Boston, 1986.
4. Yahn W Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, volume 98033, 2001.
5. Erwin Book, Dewey Val Shorre, and Steven J. Sherman. The cwic/36o system, a compiler for writing and implementing compilers. *SIGPLAN Not.*, 5(6):11–29, June 1970.
6. Francesco Di Giacomo, Mohamed Abbadi, Agostino Cortesi, Pieter Spronck, and Giuseppe Maggiore. *Building Game Scripting DSL's with the Metacasanova Metacompiler*, pages 231–242. Springer International Publishing, Cham, 2017.
7. Nicolas Ducheneaut, Nicholas Yee, Eric Nickell, and Robert J Moore. Alone together?: exploring the social dynamics of massively multiplayer online games. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 407–416. ACM, 2006.
8. Unity Game Engine. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d. com.*
9. Epic Games. Unreal engine 3. *URL: http://www. unrealtechnology. com/html/technology/ue30. shtml*, 2006.
10. Francesco Di Giacomo, Mohamed Abbadi, Agostino Cortesi, Pieter Spronck, Giulia Costantini, and Giuseppe Maggiore. High performance encapsulation and networking in casanova 2. *Entertainment Computing*, 20:25 – 41, 2017.
11. David Kågedal and Peter Fritzson. Generating a modelica compiler from natural semantics specifications. In *Summer Computer Simulation Conference*, pages 299–307, 1998.
12. Jan Willem Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.
13. Lothar Pantel and Lars C Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29. ACM, 2002.

14. George Russell, Alastair F Donaldson, and Paul Sheppard. Tackling online game development problems with a novel network scripting language. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 85–90. ACM, 2008.
15. Patrick Wyatt. The starcraft path-finding hack. `http://www.codeofhonor.com/blog/the-starcraft-path-finding-hack`, 2013.