Mogemoge : A Programming Language Based on Join Tokens

Taketoshi Nishimori, Yasushi Kuno

Graduate School of Systems Management, University of Tsukuba, Japan

nis@kjps.net, kuno@gssm.otsuka.tsukuba.ac.jp

In the video game software industry, scripting languages have been utilized to alleviate the complexity of game development. Those languages help game designers to write and modify game design quickly. However, current scripting languages do not have sufficient expressive power to describe interactions among game characters in a concise manner. This lack of expressive power makes it difficult to write and maintenance complete game rules. To overcome the problem, we are designing and developing a scripting language "Mogemoge". Mogemoge incorporates "join tokens" as primary mechanism for interaction of game characters. Join tokens are based on join calculus, with process on the original calculus substituted by tokens. In this paper, we explain language design of Mogemoge, along with its join tokens mechanism. We also discuss effectiveness of the language for game rule description.

1 Introduction

Recently, many software tools for video game development have become available. Those include game engines and development environments [8, 1, 2], which are used for development of free and commercial video games on PCs and set-top boxes (as in XBOX or PlayStation).

Scripting languages [13, 3, 5, 10] are among such tools. Scripting languages are viable alternative to traditional (low-level) programming language such as C or C++, and can provide safety and ease of writing. With those languages, non programmers (ex. game designers) can construct higher level part of game implementations such as character actions, AI controlling. As they enables game designers to implement higher level part of games without programmer's help, it has become easy to repeat trial-and-errors. More trial-and-errors makes the game more enjoyable[11].

However, most of game scripting languages are not designed for expressing complete game rules. Especially, handling of "active" objects (aka "game characters") and complex rules of their relationship are tend to be scattered among the code, leading to difficult code to write, understand and debug for non programmers.

We think that the above situation comes from the following causes:

- Game design still lacks formalization and well-known structures[11]. Consequently, programming language support for game design task is yet to be studied also.
- Concurrent programming research in general has long history, but communication models of those languages are not well suited for game description which needs lots of active characters interacting each other.

To overcome the problems, we need a language that naturally support game designers' vocabulary, with appropriate support for the description of character activities and their interactions.

The authors have previously developed a game-oriented scripting language[9] with tuple space communication model[4], and accumulated some experiences with real game development. From those experiences, it seems that tuple space is basically good, but still a bit low-level. Thus we looked for higher level communication model.

Join calculus[6] is one of such a model, in which simple string (pattern) matching in tuple space is substituted by more formal and structured join operation.

However, in join calculus, subject of join operations are active processes. This seemed too heavy-weighted for programming targeted to game activities. Therefore, we suggest "join tokens", in which processes in join calculus is substituted by passive tokens which automatically invoke associated activities.

In this paper, we propose a scripting language "Mogemoge", which incorporates join tokens mechanism and is targeted to game rule description.

The rest of the paper is organized as follows. In section 2, we briefly explain the programming language Mogemoge. In section 3, we introduce join tokens in Mogemoge. In section 4, we present a sample game program and its applicability to description of complicated game rules. Finally in section 5, conclusions and future development plans are presented.

2 Design of Mogemoge Language

Mogemoge is a prototype-based, object-oriented language like Self[12] and Dolittle[7]. The following code creates a bank account object.

```
Account = object {
  v = 0;
  deposit = method(n) { v = v + n; };
  withdraw = method(n) {
    v = v - n;
    if (v < 0) { v = 0; }
  };
  get = method() { result v; };
};
```

Mogemoge does not have "class" structure; the above code creates an ordinary object and assigns it to the global variable named "Account."

To create individual account object, one can use "`new`" operator as in the following:

```
a = new Account;
```

The "`new`" operator creates a fresh object and then copies all properties (variables and method values) from Account object.

To invoke methods on a object, dot notation as in Java or C++ is used:

```
a.deposit(100);
a.withdraw(50);
print "outstanding : " + a.get();
```

`print` is a special operation which outputs string value to standard output.

Mogemoge has four types of values:

Numeric: Represents numeric values. Numeric operators (ex. `+`, `-`, `*`, `/`) can be used for those values.

String: Represents sequence of characters. Any value can be concatenated with a string with `+` operator, resulting in appropriate string representation.

Object: An objects is a set of variables (property names and corresponding values). An object can be generated by an object literal (as in `Account` above) or a new operator.

Method: Methods are special object that can be executed by the script engine.

`Account` looks like a class, but it is simply a variable which holds an object. The `new` operator takes an object as the argument, whose copy is created and returned.

Similarly, `deposit`, `withdraw` and `get` looks like a method, but it is simply a variable which holds an method value. The `method` operator creates a method value.

An assignment stores a value to specified variable. When the specified variable does not exist, it is newly created. A 'my' modifier forces to create a local variable for a surrounding scope as in Perl. In the following, within `foo`, `a` is 1 and `b` is 2 but outside of the `foo`, `a` is 5 and `b` is 3.

```
a = 5; b = 3;
foo = method() {
  my a = 1; my b = 2;
  result method() {
    print "a = " + a;
    print "b = " + b;
  };
};
```

Note that `foo` returns an anonymous method object, using a result statement. The method can be used as in the following:

```
m = foo();  m();
```

Mogemoge also has the following features, which will be described in other papers.

- C# like delegator
- Composition (composing objects and create a new object)
- Injection (modifying an object by adding variables)
- Extraction (modifying an object by deleting variables)

In the following sections, we concentrate on join token mechanism of Mogemoge.

3 Join Tokens

Most of video game programs have a main loop which constitute the following phases:

- Update all characters
- Handle events caused in the updating phase

In the first phase, statuses of each characters are updated, representing the advance of current time by a small amount. Some events (ex. the collision between characters) may occur in the updating phase. In the second phase, those events are handled. Game rules describe complex inter-object relationships as to how those events are generated and how they should be handled.

Event handling strongly depends on character types and statuses. For example, if there are two characters which are colliding in a shooting game, following cases are possible:

- If one is an enemy's shot and another is a player, the shot will damage the player and the shot will be destroyed. But if the player is unbeatable, the shot will not be able to damage the player.
- If both are shots, they will destroy each other.
- If neither of two characters are shots, they should not overlap each other.

Programming languages suitable for game programming should support appropriate communication model which can naturally express relationships as in the above example.

Join tokens is such a communication model, which comes from join calculus[6] and tuple space[4]. **Figure 1** depicts how communications using join tokens occur.

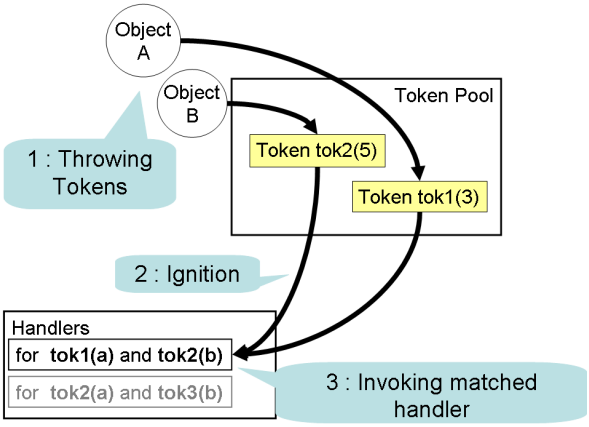

Figure 1: Join Tokens

In our model, a token is a name accompanied by a series of values (arguments). Objects can throw a token with

a name and several arguments into a place called "token pool." Handlers are special kind of code fragment that can handle tokens. Every handler has a header, which contains several token names and corresponding formal arguments.

When the special "ignition" operation is executed, the handler whose header matches tokens in the pool "fire" and execute their body part. Matched tokens are removed from the pool.

The following code throws three tokens and an ignition operation invokes a handler with two tokens of them.

```
join r1.tok1(a, b) r2.tok2(c, d) {
  print "a + c = " + (a + c);
  print "b + d = " + (b + d);
};
throw tok1(1, 2);
throw tok2(30, 40);
throw tok3("hello");
ignition;
```

The `throw` statement throws a token into the token pool. In the above example, the first `throw` statement throws a token named `tok1` into the token pool with arguments `(1,2)`. Similarly, the second `throw` statement throws `tok2(30, 40)` and the third `throw` statement throws `tok3("hello")`.

An `ignition` statement starts matching and invoking handlers. All handlers are matched by all tokens and matched handlers are invoked. In the above code, the handler is matched by `tok1(1, 2)` and `tok2(30, 40)`, and it is invoked with those tokens.

The `join` statement defines a handler which will be invoked when `tok1` and `tok2` tokens are both in the token pool. The term `r1.tok1(a, b)` means that it matches `tok1` token which has two arguments. Invoking that handler, `a` and `b` are assigned a value of corresponding arguments of a matched token. `r1` is assigned an object which have thrown the matched token.

`tok1` and `tok2` tokens are used as arguments of the invoked handler and removed from the token pool. Unused token `tok3("hello")` is left in the token pool.

Join handlers may be guarded by Boolean expressions introduced by `where`. In the following example, the handler is invoked only when the arguments of two tokens are identical.

```
join r1.tok1(a) r2.tok2(b)
     where a == b { ... };
```

Instead of removing matched tokens, it is possible to leave them in the token pool introduced by a symbol "`*`".

```
join r1.tok1(a) *r2.tok2(b) { ... }
```

Note that tokens left in the pool can match another handler, or remain in the pool until next ignition.

In contrast to `throw` statement, `dispose` statement removes a token from the token pool. The following removes `tok1` thrown by an object which executes this `dispose` statement.

```
dispose tok1;
```

Tokens are identified by its name and originating object. If an object throws tokens with the same name multiple times, the one thrown most recently remains in the pool; previous ones are overwritten.

As an example in the following, `tok(1)` is overwritten by `tok(2)` because the thrower of `tok(1)` is identical to the thrower of `tok(2)`.

```
A = object {
  m = method() {
    throw tok(1);
    throw tok(2); # overwrites tok(1)
  };
};
A.m();
```

In the following, `tok(1)` is not overwritten by `tok(2)` because the thrower of `tok(1)` is different from the thrower of `tok(2)`.

```
A = object {
  m = method() { throw tok(1); };
};
B = object {
  m = method() { throw tok(2); };
};
A.m(); B.m();
```

With overwriting mechanism, tokens in the pool act as a kind of storage cells (aka variables), and can be used to record statuses of game characters. We explain this in the next section.

To confirm existence of a token, An `exist` operator can be used. The following code confirms the existence of `tok` token whose thrower is the object executing the code.

```
if (exist tok) { ...  }
```

When the status of a character is represented by a set of tokens, `exist` operator is useful for examining statuses of the characters.

4 An Example

To evaluate Mogemoge and join tokens, we made a sample game application (**Figure 2**).

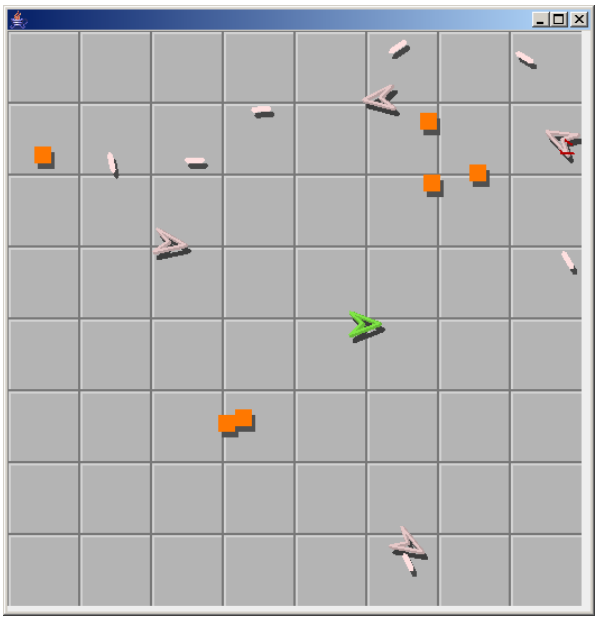

Figure 2: A Screen Shot of the Example Game

The game is a shooting game in which ships (arrowhead shapes) try to beat each other. Ships can shoot missiles (short line segments) to damage others. The player can control his ship with a keyboard. Enemies are controlled by the program. Player's purpose in the game is to destroy all enemies.

Followings are summary of game rules.

R1. Ships are controlled by a player or a program.

R2. Ships must not overlap each other.

R3. A ship can shoot missiles to damage other ships.

R4. By getting a power food, a ship becomes "unbeatable" for a while.

R5. An unbeatable ship can damage other ships by colliding against them.

R6. An unbeatable ship can destroy missiles.

R7. Ships getting a certain amount of damage are destroyed.

R8. When a ship accumulates a certain amount of damages, it is destroyed.

These rules are classified into two categories — rules that specify relationships between game characters (**R2**, **R3**, **R4**, **R5**, **R6**) and other rules (**R1**, **R7**, **R8**). Non-relationship rules (**R1**, **R7**, **R8**) can naturally be implemented as ordinary method associated with corresponding objects. However, with ordinary methods, relationship rules (**R2**, **R3**, **R4**, **R5**, **R6**) require complicated coding because two or more objects participate them. Here, our join tokens mechanism comes in.

Figure 3 shows skeleton implementation of above rules in Mogemoge (details are omitted for clarity). `x,y` and `dir` hold geometry information of ships. To determine the shot shooter, every ship has its own unique ID (stored in `id`), and all shots also record their shooters' ID in `id`.

`update` implements the main action, which is executed once for every animation frames. `is_collide` checks collision against other ships. `damage` damages the ship. Initially, `init` is invoked and `normal` token is thrown by every ship. When `make_unbeatable` is invoked, `normal` token is removed and `unbeatable` token is thrown, representing change of status for the ship.

Note that `normal` and `unbeatable` tokens describe statuses of a `Ship`. The `init` method initializes a `Ship` status as a `normal`. In `update`, `exist` operator is used to test if the ship is unbeatable, and if it is, remaining time is decreased and state is changed to `normal` when the time expires. `make_unbeatable` describes to change from "normal" to "unbeatable".

Now we turn to our four relationship rules (**R2**, **R3**, **R4**, **R5**, **R5**). The rule **R2** is about normal status ships. The rule **R3** is about ships and missiles. The rule **R4** is about ships and power foods. The rule **R5** is about a normal status ship and an unbeatable status ship. The rule **R6** is about a missile and an unbeatable status ship.

Figure 4 is the code which implements those rules. In that sample, tokens are used as statuses of game characters.

For example, the `rule R3` handler represents that if a ship (which is not unbeatable) and a missile are colliding and the ship is not a shooter of the missile, the ship is damaged by the missile and the missile is destroyed. As the missile is destroyed and `missile` token doesn't have the necessity reside in the token pool, there is not a `*` symbol on the pattern `m.missile(d)`.

By introducing join tokens, relationship rules are expressed concisely. **Figure 4** directly and declaratively represented rule descriptions summarized in this section. There are no codes to iterate on characters list or combinate characters.

5 Conclusion

In this paper, we pointed out that current scripting languages lack expressive power for complicated game interactions, and proposed "join tokens" mechanism to handle those interactions in a natural and concise manner.

We also described Mogemoge language, which incorporates join tokens as primary inter-object communication mechanism, and demonstrated its expressive power using example game application.

Mogemoge is a sequential language and it is similar to ordinary programming languages except for join tokens. In our previous research [9], we have proposed a concurrent scripting language which has the feature to describe state machines. To improve expressiveness, it is necessary to integrate those features and join tokens.

Throw-dispose scheme will be alleviated by introducing the notation of grouping or excluding tokens. It will be a useful way to composing some tokens as an abstract state machine explicitly.

Surely Mogemoge has some problems, but we confirm the possibility to express video game rules. We expect that Mogemoge will be an effective programming tool for video game software development.

References

[1] gamestudio. http://www.3dgamestudio.com/.

[2] Havok. http://www.havok.com/.

[3] Unrealscript language reference. http://unreal.epicgames.com/UnrealScript.htm.

[4] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[5] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua-an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.

[6] projet Moscova INRIA Rocquencourt. The join calculus language. http://pauillac.inria.fr/join/.

[7] Dae-Yong Kwon, Hye-Min Gil, Yong-Cheul Yeum, Seoung-Wook Yoo, Susumu Kanemune, Yasushi Kuno, and Won-Gyu Lee. Application and evaluation of object-oriented educational programming language ‘ dolittle ’ for computer science education in secondary education. *The Journal of Korean Association of Computer Education*, 7(6):1–12, 2004.

[8] Michael Lewis and Jeffrey Jacobson. Game engines in scientific research - introduction. *Communications of the ACM*, 45(1):27–31, 2002.

[9] Taketoshi Nishimori and Yasushi Kuno. An action game-oriented programming language (in Japanese). *IPSJ Transactions*, 44(SIG15), 2003.

[10] Thiadmer Riemersma. The small scripting language. In *Dr.Dobb's Journal*. CMP Media LLC, October 1999.

[11] Katie Salen and Eric Zimmerman. *Rules of Play*. MIT Press, 2002.

[12] David Ungar and Randall B. Smith. Self: the power of simplicity. *OOPSLA'87*, pages 227–242, 1987.

[13] Alex Varanese and John Romero. *Game Scripting Mastery*. Premier Press Inc., December 2002.

```
Ship = object {
  id = 0; x = 0;  y = 0;  dir = 0; timer = 0;
  init = method() {
    throw normal;   # init ship's status
  };
  update = method() {
    if (exist unbeatable) {
      timer = timer - 1;
      if (timer < 0) {
        dispose unbeatable;
        throw normal;    # change ship's status
      }
    }
    if (is_key_pressed(KEY_SPACE)) {
       m = new Missile;
       m.x = x;  m.y = y;  m.dir = dir;
       m.set_id(id);   # owned by this ship
    }
    # modifying x,y,dir to control the ship ...
  };
  make_unbeatable = method() {
    timer = 100;
    dispose normal;
    throw unbeatable;  # change ship's status
  };
  damage = method(d) {
    # increase damage
  };
  is_collide = method(o) {
    # check collision
  };
  # other methods ...
};
```

Figure 3: A Ship Code

```
# rule R2
join *s1.normal() *s2.normal()
    where s1.is_collide(s2) {
  # move s1 and s2 to prevent overlapping
};
# rule R3
join *s.normal() m.missile(d)
    where s.is_collide(m) && s.id != m.id {
  s.damage(d);
  m.destroy();
};
# rule R4
join *s.normal() p.power
    where s.is_collide(p) {
  s1.make_unbeatable();
  p.destroy();
};
# rule R5
join *s1.unbeatable() *s2.normal()
    where s1.is_collide(s2) {
  s2.damage();
};
# rule R6
join *s.unbeatable() m.missile(d)
    where s1.is_collide(s2) {
  m.destroy();
};
```

Figure 4: Rules implemented in Mogemoge