Join Tokens: A Language Mechanism for Interactive Game Programming

Taketoshi Nishimori[a], Yasushi Kuno[a]

[a] *Graduate School of Business Sciences, University of Tsukuba, Tokyo, 3-29-1 Otsuka, Bunkyo-ku, Tokyo, 112-0012, Japan*

Abstract

In the video game software industry, scripting languages have been utilized to alleviate the complexity of game development. Many of the complexty resides in managing multiple concurrent activity of game characters, especially in action games. However, current scripting languages seems to lack support for interactions among multiple concurrent activities in a sate-dependent manner. To overcome the problem, we are proposing "join token" mechanism, in which states of game characters can be expressed as tokens and interactions can be described as handlers specifying multiple tokens. For evaluation purpose, we have developed a game scripting language "Mogemoge," and have written several example games in that language. In this paper, we explain our join token mechanism, design and implementation of Mogemoge, and evaluation through an example demo game.

Keywords: video game, programming language, scripting language

1. INTRODUCTION

Recently, efforts required for developing commercial video games have significantly increased. This is because platforms for those games (PCs or game consoles such as Xbox or PlayStation) are becoming more powerful meaning that game programs are becoming larger and more complex than ever.

One way to deal with the problem is use of game-oriented scripting languages. Scripting languages enables more abstract and compact descrip-

Email addresses: nis@nisnis.jp (Taketoshi Nishimori), kuno@gssm.otsuka.tsukuba.ac.jp (Yasushi Kuno)

URL: http://www.gssm.otsuka.tsukuba.ac.jp/ (Yasushi Kuno)

tion in general, leading to shorter and comprehensive code. Game-oriented scripting languages can additionaly be equipped with language mechanisms specifically suitable for game descripton, offloading game programmers' burden further.

One of the domain in which such language mechanisms are desireble is interaction between muitiple game characters, as often seen in action games. From the game players' viewpoint, many game characters are acting concurrently, with complex interaction among them, often depending upon their states.

For example, in many shooting games, multiple missiles are concurrently moving on the game screen, and when those missiles "hit" various objects, resulting effects will be defferent depending on the kind of those objects and their states (e.g. have shield or not and such).

Managing concurrent activities in general-purpose programming languages (such as C++ or Java) is notriously difficult and complex. Moreover, when interaction is dependent on each activities' states, conditional synchronization (such as "wait" and "signal" operaitons upon conditional variables) will be required, making the situation worse.

Some of the scripting languages are adressing part of those problems.

Stackless Python[1] supports microthreads, making it feasible to assign dedicated thread to each of the game characters. It makes description of multiple concurrent activities by those characters simpler, but it does not address description problem for interaction among characters.

UnrealScript[2] supports concurrent objects called "actors" and notion of states over them (method annotated with state names are called only in corresponding states). By mapping game characters to actors and their states to actor states, natural description becomes possible. However, UnrealScript does not address the difficulty of conditional synchronization.

In this paper, we propose a new language mechanism "join token" that coordinates muitple, state-dependent concurrent activities required in game description. To assess effectiveness of this mechanism, we have also designed and implemented an experimental game-oriented scripting language "Mogemoge" which incoporates joint token as built-in synchronization mechanism. For the evaluation purpose, we have described several demo games using Mogemoge.

Concept of join token is based on join calculus[3] and Linda[4]. Join calculus models coordination of multiple concurrent tasks, and Linda models decoupling of message sender and receiver. There are several programming languages based on either of these, but none has combined both, as far as we know.

Structure of this paper is as follows: In section 2, we explain the idea and design of join token and discuss its characterstics. In section 3, we describe overview of Mogemoge language, along with its implementation. In section 4, we explain an example game described with Mogemoge and discuss effectiveness of join token. In section 5, concusion is given.

2. JOIN TOKENS: IDEA, DESIGN AND CHARACTERISTICS

As described above, join token is a language mechanism specifically targeted for description of multiple concurrent and state-dependent activities ofthen seen in action games. First, we describe the idea behind it.

Many video game programs have a main loop consisting "update all characters" phase and "handle events caused by previous update" phase. In the former phase, statuses of each characters are updated accoring to the small advance in current time; there some event (such as collision of characters) might be generated. In the latter phase, events are handled and their effects are recorded for processing in next update phase.

Some of the today's game scripting languages are based on object-orientation, because entities that must be handled within programs can naturally be mapped to objects, and this eases program description in general.

In object-orientation, behaviors (actions) are described as methods attached to one of those objects. Methods are implemented as subroutines and called from other methids (or main routine).

However, the above design largely differs from interactions in game programs:

- Interactions in games are associated with two or more characters, while methods are attatched to single object.
- Interactions are initiated when some conditions (over the associated characters) are met, while methods are invoked from some other method (under the control of the object that possess the method).
- Interaction initiations are controlled by the states of each associeted characters, while method invocations are controlled solely by the calling object.

Therefore, we have added the following new mechanism, "join token," to conventional object-orientation (figure 1).

- Each object participating in an interaction expresses its wilingness to participate by generating a "token." A token is associated with the

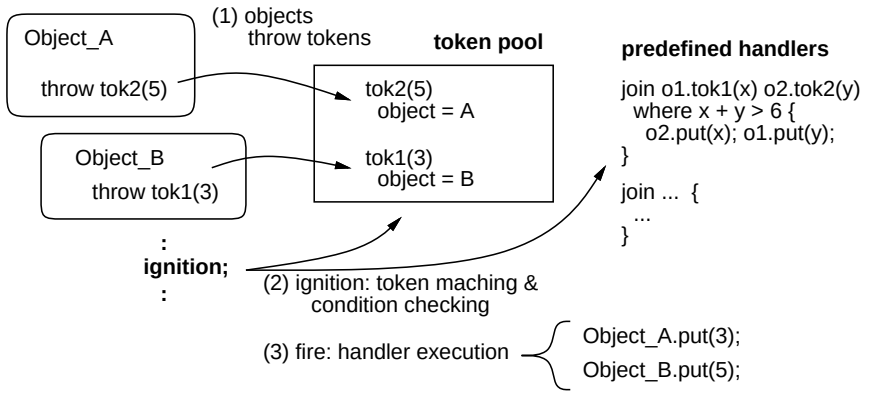

Figure 1: Idea of Join Tokens

object which geneated the token, and list of parameters specified in the code.

- Tokens are generated when methods execute "throw statements," and generated tokens are automatically put into the global "token pool."
- An interaction is described as a "join statement" that defines a "join handler" ("handler" for short). A handler specifies set of tokens which participate in the interaction, optional conditions, and body statements that are executed when the interaction occurs.
- Interactions are started when the special "ignition" statement is executed somewhere in the program; this statement corresponds to "handle event" phase noted above, and is expected to be used in the main loop. When ignition statement is executed, the token pool scans the list of defined handlers one by one, and tries to select the tokens that matches with a handler.
- When all of the handler's token specifications could be matched with existing tokens (and handler's associated condition is true, if specified), the handler "fires," and body of the handler is executed. Within the body, each token's associated objects and parameters are available. Tokens that participated in a fire are removed from the token pool unless otherwise specified.

The major benefits of the above design is that handlers are neutral to all objects and associated with the global token pool. Separation of object in-

teractions (handlers) and each objects' behavior (methos) greatly simplifies the structure of game scripts, as shown in later sections.

Some additional explanaions about the details of maching mechanisms are in order. We are not desigining a parallel programming language but a game scripting languges, in which event ordering are strictly defined and controllable.

Therefore, list of handlers are scanned in the definition (source program) order, and each handler consumes as much tokens as possible when it is considered, with maching being done in the order described in the corresponding join statement. Token maching is also done in strict orderings; older token in the pool is considered earlier. Note that a handler can fire muitiple times when there are sufficient tokens and conditions permit.

Tokens generated within the method bodies invoked from the handler bodies cannot perticipate in subsequent matching; they have to wait for the next ignition statement.

Tokens are identified with their names and the originating object. Therefore, when an object throws tokens with the same name twice (before the former one is consumed), the latter token replaces the former one (the number of arguments may vary among them). Such operations are useful when one would like to overwrite some token's arguments. Alternatively, one can withdraw a token with "dispose" statement.

As noted above, the idea of join token is based on two computational models, join calculus and Linda. Join calculus models coodination of multiple actions in a rendezvous, and corresponds to join handler in our work. Join calculus provide mechanism to synchronize multiple activity in a comprehensive way, but is mainly concerned with control synchronization. Linda is a communication model based on tokens accompanied with parameters, and corresponds to tokens in our work. Linda provides data/state exchange among concurrent activities, but does not provide convenient synchronization mechanism among multiple concurrent activities. Join token combines strong points of both models.

In the following section, we describe overall design of the Mogemoge langauge, and then show actual examples of token control statements explained above.

3. DESIGN AND IMPLEMENTATION OF MOGEMOGE LANGUAGE

As described above, Mogemoge is an experimental game scripting language equipped with join token mechanism. The purpose of developing the

language is to evaluate usefulness and descriptive potential of join token. Therefore, Mogemoge was designed to be a minimal, compact and ordinary programming language except for join token.

We have used prototype-based object-orientation as in JavaScript, Self[5] and Dolittle[6] because it can lead to compact language definition. In the same line, we have kept functionarities of Mogemoge minimal, namely: (1) object definition/creation, (2) method definition/invocation and (3) describing actions through executable statements. Below we explain (1) through (3) with small examples, and then proceed to join token-related functionalities.

The following Mogemoge 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; };
};
```

The above code creates an ordinary object and assigns it to the global variable named "Account." Within the object definition, variable assignments define and initialize the object's instance variables. Note that methods are also ordinary objects and stored in instance variables.

A "`new`" operator creates an object throgh copying.

```
a = new Account;
```

In the above code, the "`new`" operator creates a fresh object and then copies all properties (variables and their values) from Account object. Resulting object is assigned to the variable "`a`."

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 (as in Java, "`+`" acts a string concatination operation when one of its operand is a string; the other is converted to string if necessary).

Mogemoge has four types of values:

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

String: Represents sequence of characters. Any value can be concatenated with a string using a `+` operator, as explained above.

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

Method: Methods are special object that can be executed by the script engine, and described as method literals as shown above.

An assignment stores a value to the specified variable. When the variable does not exist, it is created anew. A '`my`' modifier forces creation of local variable for 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;
  };
};
```

The "`result`" statement behave just as return statements in other languages. Therefore, the method `foo` returns an anonymous method object. Method invokaction is denoteded with parenthesis.

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

Mogemoge also has the following features, which we will not describe here further.

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

Below we show token operations in syntax of Mogemoge.

A `throw` statement adds a token to the golobal token pool.

```
throw tok1(1, 2);
throw tok2(30, 40);
throw tok3("hello");
```

The first `throw` statement throws a token named `tok1` into the token pool with arguments `(1,2)`. Others are likewise.

Contrary to a `throw` statement, a `dispose` statement removes a token from the token pool. The following removes `tok1` thrown by the object executing the method code.

```
dispose tok1;
```

A `join` statement defines a handler. The following is an example of a handler definition.

```
join r1.tok1(a, b) r2.tok2(c, d) {
  print "a + c = " + (a + c);
  print "b + d = " + (b + d);
};
```

In the above example, the hanlder fires when tokens `tok1` and `tok2` tokens are both in the token pool. The term `r1.tok1(a, b)` means that it matches `tok1` token and two arguments can be extracted; when the number of actual arguments is not 2, extra valuse are discarded and `nil` valuse are used for lacking values.

When the handler is invoked, `a` and `b` represents the corresponding argument values for the matched token, and `r1` represents the object which have thrown the matched token. The term `r2.tok2(c, d)` can be read likewise. When the body of the handler is being executed, matched values can be used.

Tokens mached against a hanler are removed from the pool by default, but when a token specifier is prefixed with the symbol "`*`," the token is retaind in the pool. Following is an example.

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

Note that tokens left in the pool can be cansumed by an another handler defined below in the code, or can remain in the pool until next ignition.

Join handlers may optionally be guarded by Boolean expressions introduced by `where` clause. 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 { ... };
```

Aside from `join` statements, existense of a token can be examined by an `exist` operator, as in the following.

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

We have inplemented Mogemoge with Java and SableCC[7] compiler-compiler framework. Lexicical and syntax definition (about 200 lines of code) is translated by SableCC to Java code, which implement lexical analyzer and parser. Parser generates abstract syntax tree (AST) from the source program. Our interpreter inherits from tree walking code (also generated by SableCC) and execute program actions while traversing the tree. Total size of the Mogemoge interpreter is about 3000 lines of code, including Java and SableCC definitions.

The token pool, tokens and join handlers are implemented as data structures implemented in Java. When an ignition statement is executed, list of defined join hanlers are examined one by one, looking for mached tokens in the pool. When sufficient token for the handler is found, where clause is executed (if any), and then (if the condition was satisfied) handler body is executed. Note that where clauses and handler bodies are represented as AST data structures and stored within the handler object.

Current algorithm for token-handler maching uses simple linear search, but has not caused any perfomance problems upto around 100 handlers and 1000 tokens so far. When necessary, we could implement additional index data structures to speed up the search.

In the following section, we show an example game described in Mogemoge.

AN EXAMPLE GAME IN MOGEMOGE

To evaluate descriptiveness of join token, we have built a sample game application (**Figure 2**) in Mogemoge.

The game is a shooting game in which ships (arrow-head 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 in a single shot are destroyed.
R8. When a ship accumulates a certain amount of damages through several shots, 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,

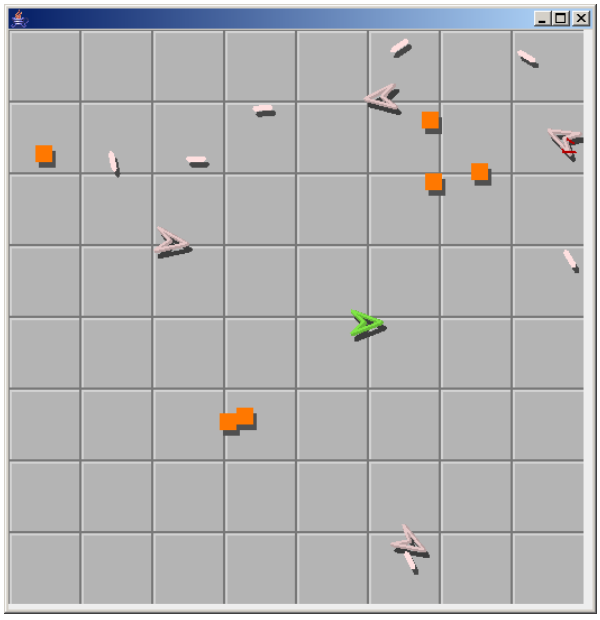

Figure 2: A Screen Shot of the Example Game

relationship rules (**R2**, **R3**, **R4**, **R5**, **R6**) require complicated coding because two or more objects participate them. Here, our join token 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 identify the shooter of a shot, 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 from main method not shown here. `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 change from "normal" to "unbeatable".

Now we turn to our five relationship rules (**R2**, **R3**, **R4**, **R5**, **R6**). 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 skelton code which implements those rules. In this sample,

```
Ship = object {
  id = 0; x = 0;  y = 0;  dir = 0; timer = 0;
  init = method() {
    throw normal;      # initialize 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)) {  # shoot a missile
       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: Souce Code for Ship Object

```
# rule R2
join *s1.normal() *s2.normal() where s1.is_collide(s2) {
  # adjust s1 and s2 coordinates 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: Source Code for Game Rule Implementation

tokens are used as statuses of game characters.

For example, the handler for `rule R3` describes 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. Note that when token specification on the handler is not prefixed by a `*` symbol, corresponding token is removed from the token pool.

Through introduction join token, relationship rules are expressed concisely. **Figure 4** directly and declaratively represents the rule described above. There are no codes to iterate on characters list or combinate characters because token matching is automatically done by the token pool.

DISCUSSION AND CONCLUSION

Game scripting languages are an effective approach in developing complex games. In the case of action games, its difficulty in development mainly resides on describing complicated interactions between multiple concurrent behavior of objects in state-dependent way.

Existing scripting languages such as Stackless Python or UnrealScript support large number of concurrent activities or object states, but they do not address the problem assosiated with state-dependent intaraction among objects well.

Many game-oritented scripting languages are appearing these days, such as GameMonkeyScript, AngelScript, and Squirrel to name a few. On those languages/libraries, problems of multiple concurrent activities or object states are addressed in various way, but still there are no outstanding approach to the problem of interaction among multiple concurrent activities.

Join token mechanism described in this paper address the problem by means of the global token pool and join handlers. This mechanism combines strong points of join calculus and Linda comoputational models.

In join token, each concurrent object can throw tokens to the pool, hereby representing its willingness to participate in an interaction. At some later point, join handlers matching with multiple tokens executes on behalf of those objects, realizing the interaction. With this framework, complex state-dependent interaction between mutiple concurrent object can be described in a comprehensive and straightforward manner.

To show the effectiveness of join token mechanism, we have designed and developed an experimental game scripting language called Mogemoge. Mogemoge is an interpreted, prototype-based object language equipped with joint token. We have developed Mogemoge using Java and SableCC (a Java-based comipler compiler framework).

For an evaluation purpose, we have described several demo action game with Mogemoge, including the one described in this paper. As the result, muiti-character shooting game could be developed straight from the game specification, resulting in simple and comprehensive code. We think this indicates effectiveness of join token mechanism for target domain (multi-character action games).

By this time, we have only developed several example games with Mogemoge. We would like to evaluate effectiveness of join token with more complex, realistic games in the future.

ACKNOWLEDGEMENT

Authors would like to thank the revewers of Entertainment Compuing, who have given us a lot of kind advices to improve the paper.

References

[1] C. Tismer, Continuations and Stackless Python, Proceedings of the 8th International Python Conference.

[2] Unrealscript language reference, http://udn.epicgames.com/Three/UnrealScriptReference.html.

[3] C. Fournet, G. Gonthier., A calculus of mobile agents, Splinger Lecture Notes in Computer Science 1119.

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

[5] D. Ungar, R. B. Smith., Self: the power of simplicity, OOPSLA'87 (1987) 227–242.

[6] D.-Y. Kwon, H.-M. Gil, Y.-C. Yeum, S.-W. Yoo, S. Kanemune, Y. Kuno, W.-G. 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) (2004) 1–12.

[7] E. Gagnon, SableCC, an object-oriented comiler framework, Master's Thesis, McGill University (1998).