Join Token-Based Event Handling: A Comprehensive Framework for Game Programming

Taketoshi Nishimori and Yasushi Kuno

Graduate School of Business Sciences, University of Tsukuba,
Tokyo, 3-29-1 Otsuka, Bunkyo-ku, Tokyo, 112-0012, Japan
nis@nisnis.jp,
kuno@gssm.otsuka.tsukuba.ac.jp
http://www.gssm.otsuka.tsukuba.ac.jp/

Abstract. In action game programming, programmers have to control multiple concurrent activities on the screen corresponding to multiple game characters. To address this difficulty, many game-oriented scripting languages have been proposed so far. However, current scripting languages seem to lack support for interactions among multiple concurrent activities in a state-dependent manner. To overcome this problem, we propose an event handling framework called "join token" in which the states of game characters can be expressed as tokens and interactions can be described as handlers specifying multiple tokens. For the purpose of evaluation, we have developed a game scripting language called "Mogemoge," and wrote several sample games in this language. In this paper, we describe experiences of using join token framework for sample games and compare the code written in Mogemoge against a code written in an existing scripting language.

Keywords: video game, programming language, event handling framework, scripting language

1 INTRODUCTION

Video game programming, especially that used for action games, has the distinguishing characteristic that programmers have to manage multiple concurrent activities on the screen corresponding to multiple game characters. For example, in many shooting games, multiple missiles are concurrently moving on the game screen, and when those missiles "hit" various objects, the resulting effects are different depending on the kind of objects and their states such as whether they have a shield or not.

Managing concurrent activities in general purpose programming languages such as C++ or Java is notoriously difficult and complex.

One way to deal with this problem is to use game-oriented scripting languages. Scripting languages can provide language mechanisms and/or frameworks to support concurrent activities of multiple game characters, so that programmers can describe the logic of the game in a more straightforward manner.

For example, Stackless Python [12][15] supports micro-threads that make it feasible to assign a dedicated thread to each of the game characters. However, this approach does not address the problem of interaction among the characters, which is the main focus of this paper.

UnrealScript[14] supports concurrent objects called "actors." In this scripting language, methods are invoked under some corresponding conditions. However, UnrealScript also does not address the problem of interaction among the characters.

To address this problem, we propose an event handling framework called "join token" that coordinates multiple, state-dependent, concurrent activities required for a game description[9]. To assess the effectiveness of this mechanism, we have designed and implemented an experimental game-oriented scripting language called "Mogemoge" that incorporates join token as a built-in coordination mechanism. For the purpose of evaluation, we have written several demo games using Mogemoge.

The concept of join token is based on join-calculus [4] and Linda[7] computational models. Join-calculus models the coordination of multiple concurrent tasks. Linda models the decoupling of the message sender and receiver. As far as we know, there are several programming languages based on either of these models, but no language has combined both of these models.

The major contribution of this paper is our interim evaluation of join token framework through two demonstration game implementations, one of which was also written in Ruby for comparison purposes. The concept of the join token framework and an overview of Mogemoge language are already described in [9]; we briefly presented these in this paper because these are necessity to understand the main point of the research.

The structure of this paper is as follows. In Section 2, we explain the idea and design of the join token framework and discuss its characteristics. In Section 3, we provide an overview of Mogemoge language, along with its implementation. In Section 4, we explain two sample games implemented in Mogemoge, and present a comparison with one of the games implemented in Ruby/Tk. In Section 5, we explain related works and discuss the strong points of join token. In Section 6, other issue including concurrency and performance are discussed, and finally the conclusion is drawn in Section 7.

2 JOIN TOKEN: AN EVENT HANDLING FRAMEWORK SUITABLE FOR GAMES

The majority of game programs and/or game scripting languages are based on object orientation, because many games are based on simulated behavior of real (or virtual) objects. In object orientation, behaviors (actions) are described as methods attached to one of those objects. Methods are implemented as subroutines and are called from other methods (or from the main routine).

However, the above design differs significantly from interactions in game programs, as follows:

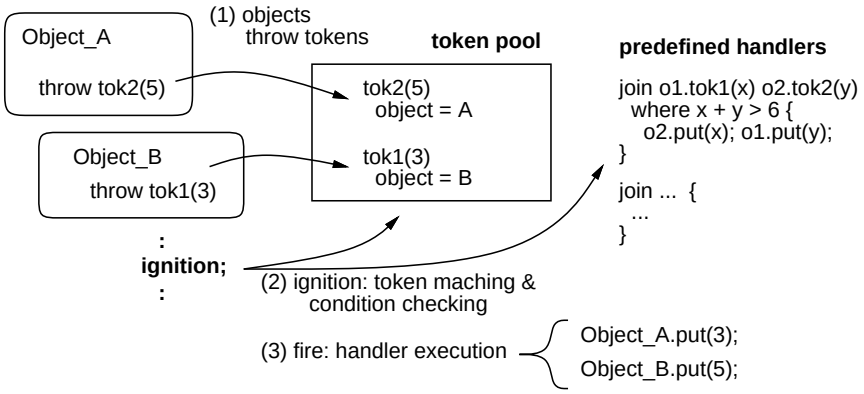

Fig. 1. Idea of join token

– Interactions in games are associated with two or more characters, while methods are attached to a single object.
– Interactions in games are initiated when some conditions are met, while methods are invoked from some other methods.
– Interactions are controlled by the states of each associated character, while method invocations are controlled solely by the calling object.

These differences make it difficult to express the programmers' (or the game designers') idea in a straightforward manner when using an object-oriented (O-O) language.

To overcome these problems, we propose the new event handling framework called "join token," as a supplementary mechanism to conventional object orientation (**Fig.1**).

– Each object participating in an interaction expresses its willingness to participate by generating a "token." A token is associated with the object that generated the token and a list of parameters specified in the code.
– Tokens are generated when methods execute "throw statements," and the generated tokens are automatically put into the global "token pool."
– An interaction (multiple object action) is described as a "join statement" that defines a "join handler" (hereafter referred to as "handler"). A handler specifies the set of tokens that participate in the interaction, the optional conditions, and the body statements that are executed when the interaction occurs.
– Interactions are started when the special "ignition" statement is executed in the program; this statement corresponds to the "handle event" phase of a game program, and is expected to be used in the main loop of the game program. When the ignition statement is executed, the token pool scans the list of defined handlers one by one, and tries to select the tokens that match with a handler.

– When all the handler's token specifications are matched with the existing tokens and the handler's associated condition is true, if specified, the handler "fires," and the body of the handler is executed. Within the body, each token's associated objects and parameters are available. The tokens that participated in a fire are removed from the token pool unless otherwise specified.

The major benefit of the above design is that handlers are neutral to all objects and are associated with the global token pool. The separation of object interactions (handlers) and each object's behavior (methods) simplifies the structure of game scripts, as shown in later sections.

Our target is not a parallel programming language, but rather a game scripting language, in which event ordering should by strictly defined and controllable. Therefore, a list of handlers is scanned in the order of their definition (source program), and each handler consumes as many tokens as possible when it is considered, with matching being performed in the order described in the corresponding join statement. Token matching is also performed strictly as per the orderings; an older token in the pool is considered earlier. Note that a handler can fire multiple times when there are sufficient tokens and when conditions permit. The tokens generated within the method bodies invoked from the handler bodies can participate in subsequent matching. Therefore, one can consume a token by a handler and immediately regenerate the same token in its body if necessary.

Tokens are identified with their names and the originating object. Therefore, when an object throws tokens with the same name twice and the first token is not consumed, the second token replaces the first one; the number of arguments may vary among these tokens. Such operations are useful when one would like to overwrite the arguments of some tokens. Alternatively, one can withdraw a token with the "`dispose`" statement.

3 THE MOGEMOGE LANGUAGE

Mogemoge is an experimental game scripting language equipped with a join token mechanism. The purpose of developing this language is to evaluate the usefulness and descriptive potential of join token. Therefore, Mogemoge was designed to be a minimal, compact and simple programming language, except for the part executing the join token mechanism.

3.1 BASIC FEATURES

We have used prototype-based object orientation similar to that used in JavaScript and Self[16], because it can lead to a compact language definition. Therefore, Mogemoge provides the syntax to create concrete objects, and those objects can be used as the prototypes from which subordinate objects (logical copies) can be created.

In the same manner, we have designed the functionalities of Mogemoge to be minimal, namely, (1) object definition/creation, (2) method definition/invocation and (3) action descriptions through executable statements. Below, we show these functionalities with the use of small code examples.

The following Mogemoge code creates a game character object:

```
Char = object {
  x = 0; y = 0;
  init_char = method(_x, _y) { x = _x; y = _y; }
  ...
};
```

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

A "`new`" operator creates an object through copying:

```
c = new Char;
```

In the above code, the "`new`" operator creates a fresh object and then copies all of the properties including the variables and their values from the Character object. The resulting object is assigned to the variable "`c`." To invoke methods on an object, the dot notation is used, similar to that used in Java or C++.

An assignment stores a value to the specified variable. When the variable does not exist, a new one is created. A "`my`" modifier forces the creation of local variables for the surrounding scope. Although the syntax was borrowed from Perl, its intention is more close to that of the local "`var`" in JavaScript.

```
foo = method() {
  my x = 1;
  result method(d) { result x; x = x + d; }
};
```

The "`result`" statement specifies the return value of the method. Therefore, the `foo` method returns an anonymous method object, which increments the value stored in `x` by `d` and returns the old value of `x`.

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

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

3.2 JOIN TOKEN FEATURE

A "`throw`" statement adds a token to the global token pool:

```
throw tok1(1, 2);
```

Conversely, a `dispose` statement removes a token from the token pool. The following statement removes `tok1` that is 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 handler fires when both tokens `tok1` and `tok2` are in the token pool. The term `r1.tok1(a, b)` means that the handler matches the `tok1` token and two arguments can be extracted; when the number of actual arguments is not 2, the extra values are discarded and `nil` values are used for the missing values.

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

The tokens matched against a handler are removed from the pool by default, but when a token specifier is prefixed with the symbol "`*`," the token is retained in the pool. Following is an example:

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

Note that the tokens left in the pool can be consumed by a sussessive handler defined in the code, or can remain in the pool until the next ignition.

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

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

Aside from `join` statements, the existence of a token can be examined by an `exist` operator, as follows:

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

3.3 IMPLEMENTATION

We have implemented Mogemoge using Java and the SableCC[6] compiler compiler framework. The lexical and syntax definition (about 160 lines of code) is translated by SableCC to Java code, which implements the lexical analyzer and the parser. The parser generates an abstract syntax tree (AST) from the source program. Our interpreter inherits from the tree walking code (also generated by SableCC) and executes the program actions while traversing the tree. The total

size of the Mogemoge interpreter is about 2400 lines of code, including the Java and the SableCC definitions.

Since Mogemoge runs on the Java Virtual Machine(JVM), it is easy to interface Mogemoge code with Java code. Actually, in our implementation, game graphics routines are written in Java and called from Mogemoge code. A special syntax is provided to declare Mogemoge-callable Java method signatures. Conversely, Java code can call Mogemoge routines and can access Mogemoge data structures (tokens and the token pool). However it is a bit difficult, because calling conventions have to be maintained.

The token pool, tokens, and join handlers are represented using Java data structures and the associated lookup code. When an ignition statement is executed, the list of defined join handlers is examined one by one, while searching for matched tokens in the pool. When a sufficient token for the handler is found, the `where` clause (if any) is executed, and then, if the condition is satisfied, the handler body is executed. Note that the `where` clauses and handler bodies are represented as AST data structures and are stored within the handler object.

The current algorithm for a token-handler match uses a simple linear search, and so far, this algorithm has not caused any performance problems with approximately 100 handlers and 1000 tokens. If necessary, we could implement additional index data structures to speed up the search.

4 EVALUATION OF THE JOIN TOKEN FRAMEWORK

We have implemented several example games using the Mogemoge language and the join token framework. In the following sections, we describe two such games and discuss our model. In addition, we have implemented one of the games using an existing programming language (namely, Ruby with Tk graphics) and compared the resulting code with the Mogemoge version.

4.1 "BALOON" GAME

"Balloon" is a simple shooting game written in Mogemoge; the number of lines of code is 357. Its screenshots are shown in **Fig.2**. In this game, balloons with hanging bombs come down from the sky. The player controls a battery (at the left-bottom corner of the game screen), and shoots/explodes missiles to destroy the bombs, so that bombs do not hit buildings at the bottom of the screen.

One missile can destroy only one balloon or bomb when the missile hits them directly. Alternatively, explosion of the missile can destroy multiple balloons and bombs that fall within the explosion area.

Rules of this game are summarized as follows:

R1. A battery's direction is controlled by the player.
R2. The player can shoot/explode a missile by pressing/releasing a key.
R3. A balloon falls slowly from the top of screen. A bomb is bound to the tip of a string hang from the balloon.

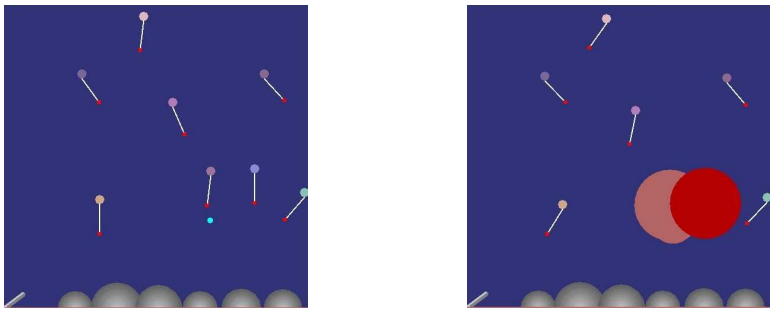

Fig. 2. Screenshot of "Balloon" Game

R4. A missile can destroy a balloon or bomb *without an explosion*.
R5. If either a balloon or a bomb is destroyed while connected, the remaining one continues to fall. A bomb falls faster than a balloon.
R6. An explosion of a missile/bomb can destroy balloons.
R7. An explosion of a missile/bomb can explode bombs.
R8. A bomb and an explosion can destroy a building at the bottom of the game screen.
R9. An explosion is not destroyed by any objects except for a building. An explosion cannot destroy two or more buildings.

These rules are classified into two categories: rules that specify relationships (or interactions) between game characters and other rules. Relationship rules are **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, and **R9**. Non-relationship rules are **R1**, **R2**. Non-relationship rules can be implemented as ordinary methods associated with corresponding objects in a straightforward manner. However, when ordinary method are used, some elaborated coding will be required to implement relationship rules, because two or more objects are involved with these rules.

With our join token framework, these relationship rules can be represented as one or more join handlers in a clear and straightforward manner.

The following code initializes a balloon and a bomb:

```
Balloon = object {
  init = method( x ) {     # a method which initializes a balloon
      bomb = new Bomb; bomb.init( x, 0 ); # create a bomb
      throw balloon(bomb); # throws a token with a bomb
  }
};
Bomb = object {
  init = method( x, y ) {  # a method which initializes a bomb
    throw bomb;            # throws a bomb token
  }
};
```

When a balloon is created, it also creates a `bomb` and a `balloon` token associated with itself (as a throwing object) and the bomb (as an argument).

Similarly, other objects such as a missile or an explosion throws a token named `missile` or `explosion` (this time with no arguments) correspondingly, at the time of creation.

The rules related to collisions/explosions (**R4**, **R5**, **R6**, **R7**, **R8**, **R9**) can be implemented with the following handler code:

```
# rule R4+R5: A missile destroys a balloon.
join m.missile() bln.balloon(bomb) where m.is_collided( bln ) {
  m.destroy(); bln.destroy();  # destroys the missile and the balloon
  bomb.set_vel( 0, 2 );        # add falling velocity to the bomb
}
# rule R4(+R5): A missile destroys a bomb.
join m.missile() b.bomb() where m.is_collided( b ) {
  m.destroy(); b.destroy();    # destroy the missile and the bomb
}
# rule R6(+R5): An explosion destroys a balloon.
join *e.explosion() bln.balloon(bomb) where e.is_collided( bln ) {
  bln.destroy();               # destroys a balloon
  bomb.set_vel( 0, 2 );        # add falling velocity to the bomb
}
# rule R7(+R5): An explosion explodes a bomb.
join *e.explosion() b.bomb() where e.is_collided( b ) {
  b.destroy();
  e = new Explosion;           # an explosion takes place
  e.init( b.x, b.y, 1.2 );     # initialize position and size
}
# rule R8: A bomb destroys a building.
join b.bomb() bld.building() where b.is_collided(bld) {
  bld.destroy(); b.destroy();  # destroys the building and the bomb
}
# rule R8+R9: An explosion destroys a building.
join e.explosion() bld.building() where e.is_collided(bld) {
  bld.destroy();               # destroys the building
}
```

The handlers corresponding to rules **R6** and **R7** use the `*` symbol to retain `explosion` tokens in the pool for some duration, so that they can destroy multiple balloons and bombs. The handler corresponding to the rule **R9** (the last handler) is an exception; an `explosion` token is deleted when an explosion collides with a building.

The binding rule, **R3**, is implemented by the following handler code:

```
# rule R3: a bomb is bound to a balloon.
join *bln.balloon(child) *b.bomb() where child == b {
    b.set_pos( bln.tipx, bln.tipy );
};
```

If there is a pair of `balloon` token and `bomb` token such that an argument of the `balloon` is the object associated with the `bomb` token, this means that

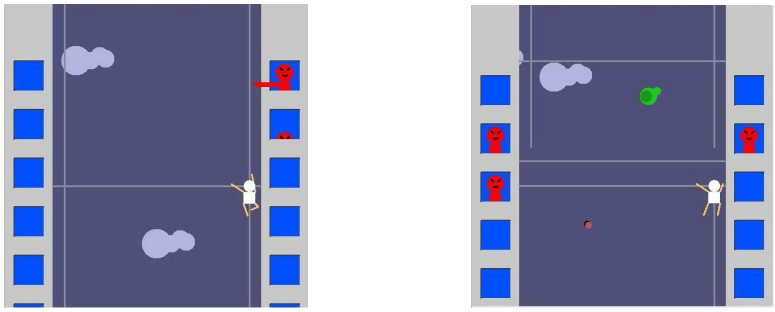

Fig. 3. Screenshot of "Descender" Game

the corresponding bomb is bound to the corresponding balloon. Therefore, the position of a bomb is set to the tip of the string of a balloon (variables `tipx` and `tipy` of a balloon means the position of the tip of the string). This handler is intended to fire many times to continually adjust the positions of bombs. So, `*` symbols are used to retain the corresponding tokens in the pool

The rule **R5** is implemented as a supplementary code in the handler for rules **R4**, **R6**, and **R7**. When a balloon is hit by a missile, the falling velocity of the remaining bomb is increased. However, when a bomb is hit by a missile, the speed of the corresponding balloon does not change, so no action is required.

As shown in the above listings, our handlers correspond to game rules in a fairly straightforward manner, and the code described in the handler bodies are simple and readable, suggesting the usefulness of our join token framework.

4.2 "DESCENDER" GAME

"Descender" (**Fig.3**) is an action game that is more complicated than the one described in the previous section; the number of lines of code is 847. The aim of the player in this game is to descend infinitely along the wall of two buildings using horizontal and vertical ropes. The player controls where to stretch ropes and his movement along them. Birds drops bombs, and building inhabitants occasionally cut vertical ropes. If a bomb hits the player or an inhabitant cuts the rope that the player is currently hanging on, the game is over.

Rules of this game are summarized as follows:

R1. The player can move along a horizontal or vertical rope from his current position.
R2. When the player is holding the bottom of a vertical rope, or there is no vertical rope under the player, he can extend a vertical rope to the bottom of the screen.
R3. All objects are scrolled upward when the player descends (the vertical coordinate of the player is fixed).
R4. Clouds are scrolled up more slowly than other scrolling characters; they are for visual decoration and have no effect on other objects.

R5. When the player is holding a vertical rope and there is no horizontal rope around him, he can stretch a horizontal rope.
R6. Birds flying in the air occasionally drop bombs.
R7. Inhabitants in some of the windows occasionally cut the vertical rope in front of them.
R8. If a bomb hits the player, the player falls and the game is over.
R9. If an inhabitant cuts the rope that the player is currently holding, the player falls and the game is over.
R10. A vertical rope extends to the bottom infinitely until an inhabitant cuts it.

All of the above rules except **R6** are implemented with join handlers.

In this game, a player is in one of three modes: descending mode (holding a vertical rope), horizontal moving mode (holding a horizontal rope), and falling mode (the game is over). A player's action in each mode is implemented as a corresponding method, and a variable named `update` stores the currently effective mode, as follows:

```
Player = object {
  update_descending = method() { ... }
  update_moving_horizontally = method() { ... }
  update_falling_horizontally = method() { ... }
  update = update_descending;   # the initial mode is descending.
}
```

The following is the descending mode method:

```
update_descending = method() {
  if ( guiKeyPressed( KEY_DOWN ) ) {
    throw cmd_descend;
  }
  if ( guiKeyOn( KEY_LEFT ) ) {
    throw cmd_shoot_hrope;
  } elif ( guiKeyOn( KEY_RIGHT ) ) {
    throw cmd_shoot_hrope;
  }
  is_left_side = ( x == LEFT_PLAYER_X );
  if (guiKeyPressed( KEY_RIGHT ) and is_left_side) {
    throw cmd_go_side;
  } elif (guiKeyPressed( KEY_LEFT ) and not is_left_side) {
    throw cmd_go_side;
  }
};
```

The actions the player can perform in a descending mode are to descend, to stretch a horizontal rope, and to switch to horizontal moving mode. These actions are expressed by throwing either a `cmd_descend`, `cmd_shoot_hrope`, or `cmd_go_side` token. For example, a `cmd_descend` token (thrown when the down arrow key is pressed) is handled by the following two handlers:

```
join p.cmd_descend *r.v_rope where r.is_on( p.x, p.y ) {
  d = min( r.by - p.y, 2 );
  if ( d > 0 ) {
    throw scroll( d );
    p.anim_descend();
  } else {
    r.extend_to_bottom();
  }
};
join p.cmd_descend { };
```

A `v_rope` token is thrown by a vertical rope at its creation time and remains in the pool as long as the rope is available (designated by a `*` symbol).

A body of the first handler is executed when there is a `cmd_descend` token and a player is on a vertical rope. This handler implements the rules **R1** and **R2**. When the vertical rope has some margin below the player to descend (**R1**), all objects are scrolled up (as the player descends) Otherwise, the vertical rope is extended to the bottom of the screen(**R2**). Note that the `scroll` token is thrown when the player is descending, as explained shortly.

The second handler has an empty body; it simply consumes a `cmd_descend` token when it is not processed by the first handler, e.g., either the player is not descending or the player is not on a vertical rope.

Other command tokens thrown in a descending mode are implemented similarly. All command tokens thrown at a certain animation frame are handled within the frame, and are consumed by an empty handler when they are not effective in that frame.

The game screen scrolls up according to the player's descending action (**R3**). Therefore, all game characters except for the player update their vertical coordinate. As shown above, the handler of rule **R1** throws `scroll` token when the player is descending, and this token is handled by the following handlers (the argument of a `scroll` token represents the number of pixels to scroll):

```
join *any.scroll(d) *o.bg_object {
  o.y = o.y - d;
  if ( o.y < SCROLL_OUT_LIMIT_Y ) { o.destroy(); }
};
join *any.scroll(d) *r.v_rope {
  if ( r.y > 0 or d > 0 ) { #
    r.y = r.y - d;
    if ( r.y < 0 ) { r.y = 0; }
  }
  if ( r.by < HEIGHT ) {
    r.by = r.by - d;
    if ( r.by < 0 ) { r.destroy(); }
  }
};
join *any.scroll(d) *c.cloud {
  c.y = c.y - d / 2.0;
```

```
};
join any.scroll { };
```

Note that the `scroll` token is thrown by the handler that consumes the token `cmd_descend` and is processed by another handlers in the same ignition. Therefore, handlers that processes `scroll` should be placed below the handler that throws `scroll`.

All objects that scroll but have no specific action while scrolling throw a `bg_object` token at initialization; they simply move upward and destroy themselves when they go out of the screen. This is implemented by the first handler.

The second handler is for a vertical rope. A vertical rope, whose top and bottom vertical coordinates are held by `y` and `by` correspondingly, behaves a little differently. As the rule **R10** states, the bottom of a vertical rope does not move upward if its bottom is at the bottom of the game screen.

The third handler implements a cloud that goes slowly (at half the speed of other scrolling objects) up the screen (**R4**). When a cloud goes out of the screen, it resets its position to the bottom to "reuse" itself (this behavior is described in the cloud object and is not shown here).

The fourth handler is defined at the end of handlers processing the `scroll` token to consume a `scroll` token when its job is done, as in the other command tokens.

The rule **R5** is implemented by the following handlers:

```
join p.cmd_shoot_hrope *r.h_rope_flying
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope *r.h_rope
      where abs( p.y - r.y ) < BREADTH_TO_CHANGE_ROPE {
};
join p.cmd_shoot_hrope {
  hr = new HorizontalRope;
  if ( p.x == LEFT_PLAYER_X ) {
    hr.init( p.y, false );
    p.anim_shoot_hrope( false );
  } else {
    hr.init( p.y, true );
    p.anim_shoot_hrope( true );
  }
};
join p.cmd_shoot_hrope { };
```

Note that a horizontal rope throws either a `h_rope` token (on which a player can hang) or a `h_rope_flying` token (on which player cannot hang on because it is not completely stretched between buildings).

A horizontal rope can be stretched only when there is no other horizontal rope nearby (**R5**). This behavior is expressed by the first 2 handlers. If there is

any horizontal rope within `BREADTH_TO_CHANGE_ROPE` pixels, a `cmd_shoot_hrope` token is simply deleted so that no further action occurs.

The third handler stretches a horizontal rope leftwards or rightwards according to the current position of the player. Note that this hander specifies only one token. Its role is to execute the stretching action when the `cmd_shoot_hrope` token was not removed by the previous handlers.

The fourth handler removes the token when its task is done as in the other handlers that process command tokens.

An inhabitant tries to cut a vertical rope (the rule **R7**). An inhabitant consists of two objects: the inhabitant itself and his arm. An arm object throws a `cmd_cut_rope` token if it has fully extended from its body, and this token is processed by the following handlers (method `can_cut` checks if the arm can actually cut the rope):

```
join a.cmd_cut_rope *r.v_rope where a.can_cut( r ) {
    r.cut( a.y );
};
join a.cmd_cut_rope { };
```

Method `cut` actually cuts the rope and throws a `cut_vrope` token with two arguments, which are the top and bottom y coordinate of the cut part. This token is processed by the handlers corresponding to **R9**, which simply checks if the player should fall and changes its state accordingly:

```
join *p.player rope.cut_vrope( by, cut_y )
    where p.x == rope.x
      and rope.y <= p.y and p.y <= by
      and cut_y < p.y {
    p.update = p.update_falling;
};
join any.cut_vrope { };
```

The rule **R8** is implemented similarly:

```
join *p.player o.bitch where p.is_collided( o ) {
    p.update = p.update_falling;   # create an effect object....
};
```

Although the "Descender" game has some complexity, the methods of the game objects are simple and the join handlers concisely express the corresponding rules in a straightforward manner.

4.3 COMPARISON WITH RUBY

As an evaluation, we have implemented our "Balloon" game also in Ruby (precisely Ruby/Tk; Tk library is used for graphics and input handling). In this section, we present the comparison between a join token-based code in Mogemoge and an ordinary O-O code in Ruby.

The resulting game was mostly identical except for the speed and/or look and feel owing to differences in graphics library. The numbers of lines of code is 357 for the Mogemoge version and 436 for the Ruby version. The object definitions are mostly similar for both the versions, but there are large differences in the event handling part.

In this game, most of the interactions among characters are collisions, e.g., an interaction occurs when two characters collide with each other. Therefore, we can factor out collision detection onto a single iteration method (a kind of coroutine in Ruby) as follows:

```
def check_collision( c1, c2 )
    o1s = $obj_list.find_all { |o| o.kind_of? c1 }
    o2s = $obj_list.find_all { |o| o.kind_of? c2 }
    o1s.each do |o1|
        o2s.each do |o2|
            yield o1, o2 if o1 != o2 && o1.is_collided( o2 )
        end
    end
end
```

With the help of a `check_collision` method, an interaction handling code can be written as follows:

```
def check_collision_all
    check_collision( Explosion, Bomb ) do |e,b|
        b.destroy()
        Explosion.new( b.x, b.y, 1.2 )
    end
    check_collision( Explosion, Balloon ) do |e,bln|
        bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
        bln.destroy
    end
    check_collision( Missile, Balloon ) do |m,bln|
        m.destroy; bln.destroy
        bln.bomb.set_vel( 0, 2 ) if !bln.bomb.nil?
    end
    check_collision( Missile, Bomb ) do |m,b|
        m.destroy; b.destroy
    end
    check_collision( Building, Bomb ) do |bld,b|
        bld.destroy; b.destroy
    end
    check_collision( Explosion, Building ) do |e,bld|
        if e.active then e.active = false; bld.destroy end
    end
end
```

Each of the calls to `check_collision` corresponds to handlers in the Mogemoge version, but there are the following differences:

- The kinds of characters participating in each interaction are explicitly described as class names, but their roles in the interaction are not shown; they are expressed as token names in Mogemoge.
- Additional objects participating in the interaction have to be stored in and extracted from the participating object; they are expressed as token arguments in Mogemoge.
- Checks for condition prior to actions are embedded in the code; they are represented as "where" clauses in Mogemoge.
- Unavailability of an object for multiple interactions have to be managed by the code through flags (`active` property in the above code); they are automatically managed by token semantics and `*` symbols in Mogemoge.
- The above Ruby code does not address rule **R3** because it is not a collision. We had to make a balloon and a corresponding bomb to refer to each other via their reference variables and had to maintain this relation manually, as in the following code, in the `destroy` method:

```
def destroy  # a balloon must not refer to a destroyed bomb
  @parent.bomb = nil if !@parent.nil?
end
```

  Using a join token, such references were not necessary and **R3** could be described in a straightforward manner within the handler.

In more complex games with many interactions other than collisions (as in the "Descender" game), the complexity of the Ruby code will increase to a great extent.

In addition, the above code runs nested loops for every combinations of interacting characters for clarity; if the performance becomes a problem, we will have to merge some of the loops, further decreasing the readability of the code. In the case of Mogemoge, we can implement various speedup techniques as necessary without affecting the existing code.

5 RELATED WORKS

There are many aspects in our join token framework, so we shall examine the related works with respect to each of them.

game scripting languages Since our goal is to ease game programming, we first examine the related works that treat game scripting languages. As noted previously, an action game programmer has to control multiple concurrent activities of game characters, along with their interaction in a state-dependent manner.

Micro-threads of Stackless Python [12], [15] allows assigning a dedicated thread to each of the game character objects, so that those objects seemingly act autonomously and concurrently; this view is very natural for game designers. Several websites including [2] recommend this style of game scripting.

Yet, another awkwardness of game programming is that each of the characters may have their own state, and they interact with each other in a state-dependent manner. Some of the game scripting languages, including UnrealScript[14] provides a notion of state; in such languages, game programmers can explicitly describe states in their code. However, "interaction" poses another difficulty, because two or more characters (with their own states) are involved in an interaction.

coordination models From the above discussion, it seems necessary to introduce some coordination model to the game scripting language in order to ease the description of interactions among concurrent activities (game characters).

Linda[7] is a coordination model that uses a "tuple space" as a communication media among concurrent activities. In Linda, both the sender and the receiver of a message (a "tuple" in Linda terminology) are separated in time (at which timing) and space (at which portion of the code). This relieves the programmer from the awkward control of details. However, the demerit of Linda is that the coordination is not symmetric and a bit too low-level; the sender simply emits its tuple, and the receiver must actively select tuples matching its needs.

Join-calculus[4] is yet another coordination model in which the atomic join handler of two or more concurrent activities can be specified. The merit of join-calculus is its high level description and symmetry. On the other hand, the target of the coordination is the thread itself and the two activities are tightly coupled at the join handler; loose coupling of Linda will be more desirable in this respect.

Therefore, we have combined the advantages of both the models and designed a join token framework. Tokens and token pool corresponds to tuples and tuple space in Linda, respectively. A join handler was derived from join-calculus, although our handlers join tokens, not threads. Moreover, we have associated an originating object to each of the tokens in order to ease the object-orientation style of programming, which is common in game programming.

reactive/event programming Join token can be viewed a kind of reactive and/or event-based programming, which has a long series of history.

First, rule engines, long used for expert systems, have the facility to gather multiple facts (similar to tuples in Linda) and invoke rules when matches are found. Moreover, recent rule engines such as Jess [5], [8] allows Java objects to be used as facts: thus, it can be used as a coordination mechanism for game program written in Java. However, such code will be awkward to write, because every coordination activity must be converted to Jess API calls. Alternatively, it is quite possible to use Jess or a similar rule engine as an implementation device for token pool and take advantage of the efficient Rete[3] algorithm built into it. We will revisit this topic later.

Second, Dynamic aspect-oriented programming (AOP) as in [1], [10] makes it possible to insert join points to existing class code at runtime. It could be used to install callbacks (join points) when an object becomes ready for interaction and

would like to wait for one of the other objects to express willingness to participate in an interaction. However, this approach is similar to a thread-based join such as in join-calculus, with its drawbacks, as noted previously. It might also be too general and powerful; a more domain-specific solution would be desirable.

Third, data binding as seen in JavaFX[11] and reactive programming[13] can trigger events when the values of some variables have changed, and appropriate action can be specified. Their major usage for now is to reflect values of some portions to other parts of the system (e.g., user interface components or accompanying objects or so on), but a more flexible setting (for game logic programming) is also possible. However, such customizations might be awkward. Therefore, they might be used as back-end mechanisms to implement join tokens, as in the rule engine case.

6 DISCUSSION

In this section, we discuss the various aspects of join token frameworks and discuss their related issues.

concurrency issue As noted in Section 2, the join token framework described in this paper is targeted to game scripting, in which event ordering and processing order should be strictly controlled by the programmer. Therefore, true concurrency and the resulting non-determinism are intentionally excluded. Perhaps, thanks to this strict ordering property, we got little surprises when debugging the join token-based code.

However, Linda and join-calculus, from which the join token was derived, are actually concurrency coordination models. Therefore, the join token model might also be useful for a true concurrent setup also. We might encounter more "surprises" with such setup, and might require additional coordination (order controlling) mechanisms. We would like to investigate this issue in the future.

restriction regarding token overwriting As explained in Section 2, each object can throw multiple tokens with different names into the pool, but can have only one token with a specific name, because the latter throw statement with identical name replaces the previous one. Although this may pose some restriction on the usage of tokens, we have chosen this condition for clarity and simplicity; we have felt no inconvenience so far.

applicability to more complex games The games we have implemented with Mogemoge and join token so far are fairly simple and small ones, so their applicability to more complex (commercial-scale) games is not yet known.

However, we note that token names are hard-coded in the source code and cannot be changed at runtime. Therefore, although the token pool looks like a global chaos, it is not so in fact; logically, there are many small pools for each distinct token names. When developing large scale games, a token naming convention can be used to safeguard against interference among program modules.

performance issue As noted above, we have only implemented small games using the join token, and so have not encountered any performance problems so far. We expect that this situation might change in the case of larger games.

However, we note that current video games use the majority of their CPU cycles in 3D high resolution graphics, so we guess that CPU cycles used for game logic computation will be negligible even on fairly complex games.

When dealing with the computational complexity of token matching, given that tokens with different names are totally distinct, the number of tokens and handlers with the same name matters. In a naive implementation (which we use for current Mogemoge implementation), with M handlers and N tokens for a specific token name, the computational complexity will be $O(MN)$.

If this becomes a problem, we could incorporate a clever algorithm such as Rete[3]. However, since the Rete algorithm caches the outcome of Boolean guard expressions ("`where`" clauses in join tokens), we need a guarantee that the value of guard expressions does not change without notice. One way to achieve this might be to restrict the guard expressions to use only local values (handler associated objects and their instance variables). We consider a detailed analysis as our future work.

7 CONCLUSION

Game scripting languages are an effective approach to develop complex games. In the case of action games, the difficulty in development mainly resides in describing complicated interactions among the multiple concurrent behaviors of objects in a state-dependent way.

The join token mechanism that was described in this paper addresses this problem by means of the global token pool and join handlers. This mechanism combines the advantages of the join-calculus and the Linda computational models.

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 join token. We have developed Mogemoge using Java and SableCC (a Java-based compiler-compiler framework).

For the purpose of evaluation, we have implemented several demo action games with Mogemoge, including the two described in this paper. We have also compared Mogemoge against an ordinary scripting language through experiments. As a result, the Mogemoge code could be easily derived from the game rules and is comprehensive in general.

At present, we have developed only a few simple sample games with Mogemoge. We would like to evaluate the effectiveness of join token in more complex, realistic games in the future.

ACKNOWLEDGEMENT

The authors would like to thank the reviewers of Software Language Engineering Conference 2011 for their helpful suggestions to improve this paper.

IMPLEMENTATION STATUS AND AVAILABILITY

The implementation of Mogemoge and its sample programs are available at the web site at `http://www.nisnis.jp/mogemoge/`.

References

1. J. Bornér. What are the key issues for commercial aop use: how does aspectwerkz address them? *3rd International Conference on Aspect-Oriented Software Development*, pages 5–6, 2004.
2. Michael Dorf. Need high levels of concurrency? Try stackless Python, July 2010. http://www.learncomputer.com/stackless-python/.
3. C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern matching problem. *Artifical Intelligence*, 19(1):17–37, 1982.
4. Cedric Fournet and Georges Gonthier. *A Calculus of Mobile Agents*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 1996.
5. E. Friedman-Hill. *Jess in Action: Rule-Based Systems in Java*. Manning Publications Co., Greenwich, CT, 2003.
6. Etinne Gagnon. SableCC, an object-oriented comiler framework, 1998. Master's Thesis, McGill University.
7. David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
8. Jess: the rule engine for the Java platform. http://www.jessrules.com/.
9. Taketoshi Nishimori and Yasushi Kuno. Join token: A language mechanism for interactive game programming. *under submission*, 2011.
10. Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: Efficient dynamic weaving for java. *2nd international conference on Aspect-Oriented Software Development*, pages 100–109, 2003.
11. Václav Slovàcek, Mirosav Macik, and Martin Klíma. Development framework for pervasive computing applications. *SIGACCESS NEWSLETTER*, 95:17–29, 2009.
12. Stackless Python. http://www.stackless.com/.
13. Jean-Ferdy Susini. The reactive programming approach on top of Java/J2ME. *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 227–236, 2006.
14. Tim Sweeney. UnrealScript language reference. http://udn.epicgames.com/Three/UnrealScriptReference.html.
15. C. Tismer. Continuations and Stackless Python. *Proceedings of the 8th International Python Conference*, 2000.
16. David Ungar and Randall B. Smith. Self: the power of simplicity. *OOPSLA'87*, pages 227–242, 1987.