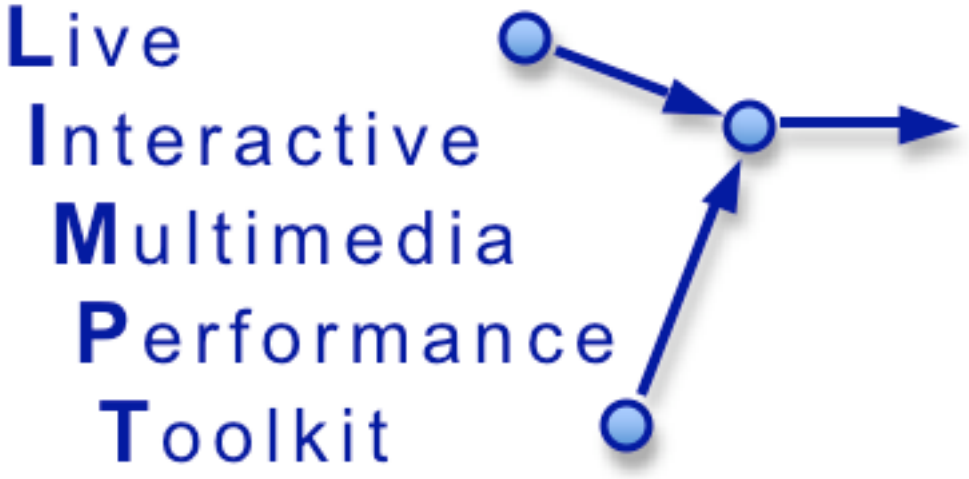
eMerge performance scripting

Version 1.33 – 24th August 2010



Ian Willcock

ian@willcock.org

Contents

Introduction

The eMerge system is a networked system that tracks performance events in real-time and is then able to issue commands to performers based upon a rule-set which has been created before the performance.

The eMerge scripting language, which controls the decision making and cue-issuing process, is based on simple text-based definitions of rules, events and commands. These are written by the user(s) in everyday language, and are then interpreted by the system. When you enter a rule or a command, the system checks that what you have entered contains the information it needs. It then either stores the information if it part of performance preparation, or it carries out the requested action(s) if it is a command for immediate execution.

Language Overview

The eMerge system

The eMerge system connects live performers and other system-level sensors to a central server. The system has two main modes – a pre-performance, preparation state and an active, performance state.

In the preparation state, the system configuration is defined and checked and the rule structures that will determine how the performance will unfold are entered and edited. In performance, the server stores information about what is happening and continually compares what has happened against a stored set of circumstantial directives – e.g. if such and such an event occurs, carry out this action.

Scripting language syntax

The basic structure of the eMerge scripting language is the *Rule*. These are entered before a performance and are stored in the system. They define what the system should do during a performance when a particular set of events occur. Rules can be active or inactive – so that responses to sets of events can change as a performance progresses.

When the triggering events occur for a rule, its command(s) are executed and the triggering events are cleared from the system's memory (so that the rule does not rigger again straight away). The rule itself however remains active unless it specifically makes itself inactive through one of its commands.

The other type of input is a *Command*. When a command is entered on its own (as opposed to when it is a part of a rule – see below), the system will carry out the requested action immediately. Commands can be entered at any time - while the system is in preparation mode or during a live performance.

Commands can cause a wide variety of actions to be carried out, these include causing information to be sent to one or more performers, or changes to the internal storage and operation of the emerge system itself.

Rules

Rules determine how the system operates when it is running a performance. A rule has two parts: events and commands. If an event happens during a performance, the command will be issued: -

If
ian says “boo”
Then
say “go” to tom

Rules can be triggered by any one of a number of events happening, and can carry out more than one command: -

If
Ian says “boo” Jane says “boo” tom says “boo” sonja says “boo”
Then
say “go” to sven show “start playing section four” to craig

Here, if any one of the named says (types) “boo” then the commands will be issued.

The events needed to trigger a rule can be also combined using ‘and’: -

If
jane says “boo”
and
Section three is active
Then
say “go” to sven show “start playing section four” to craig start section four

Events

An event specification has three parts; a *reference* to a performer or other system object, a *description* of something they might do – their *state* - and the *test* for the event to be judged as having happened

Here are some events: -

Object reference	test	What
nick	says	“stop”
rule SysName_RULE_2	is	active
performance time	is more than	10 minutes
zone view1 x	Is less than	3.5

The reference part of an event must be a word or phrase that refers to something the system knows about and holds information about (see the detailed section on referring for more information).

Things the system knows about include:

performers	zones
rules	sections
the performance	

There are four tests that may be applied to judge if the event has happened. The user can use different words and symbols to specify them. Here are just some ways of writing the four tests: -

equals	does not equal	is higher than	is lower than
is	isn’t	is more than	is less than
=	≠	>	<

Commands

Commands make the system do something. They may be triggered by events happening or they may be entered directly by a user at any time – even when the system is not running a performance.

Commands have two or three parts; an *action* to be carried out, a *reference* to a performer or other system object which is the target for that action and, where required, a *description* of what is to be done or sent to that target.

Here are some commands: -

Action	Object reference	What
start	performance	
say (to)	Hannah	“start singing”
set	rule 3	INACTIVE

The reference part of a command must follow the same rules as for events – it must be a word or phrase that refers to something the system knows about and holds information about.

The action definitions that can be used depend partly on what the system and its components (performers, editors, sensors etc.) can do. Some of the words that will always be understood by the system are: -

say	set	show	play
give a	start	stop	
delete	get	make	

Referring to Things or Objects

When a rule or command is entered into the eMerge system, the user has to provide enough information for the system to be able to identify exactly which information should be examined or sent – depending whether the reference occurs in a rule or a command.

The system works this out by examining the object reference you provide and (particularly for events) the test you specify. If a word is not one of the built-in system words, it is assumed to bee the name of an object. For rules, sections, and zones you need to specify its type: -

```
zone left_space
section introduction
rule SysName_RULE_21
```

For performers, you just need to use the name they are logged in as: -

```
jane
ian
```

Object names cannot contain spaces or punctuation marks.

Describing States or Values

There are four different ways of describing what value should be tested or sent to a system component.

generalised descriptions	ANYTHING, NOTHING, EMPTY, ACTIVE, INACTIVE
absolute descriptions	“hello”, 125, “F# 3”, 2 minutes
references to data	image “symbol1”, sound “low buzz”, midi “riff”, movie “gesture 7”
references to objects	zone upstage, rule SysName_RULE_21, performance time

Events

The eMerge system is able to keep track of, and respond to, a large number of different sorts of events. When specifying events in rules, there are often additional pieces of information about the events that can be included in rule definitions. The nature of an event is specified in rules by a combination of the choice of 'action' word and information type. Events may be associated with a particular performer – or with a system-level input device such as a pressure pad or ambient light level sensor.

Identifying event sources

Performers and the system-level objects (zones, sections and rules) are identified by name.

The ANYONE and NOONE keywords are also allowed.

Examples

(if) Jane gives a mouse signal

(if) ANYONE gives a mouse signal

System objects are referred to by name.

Examples

(if) performance time is more than 5 minutes

(if) section coda time is 2 minutes

(if) rule SysName_RULE_3 is ACTIVE

(if) zone myZone x-position is 2.0

After the source identification, an event definition must contain information about what sort of input and what value that input might have to be considered meaningful. The types of input that are currently implemented include; *signals* (of various types), *text* and *midi*.

Signals

Signal events are simple cues generated by the user or device interfacing with a client. They could include such things as mouse clicks, a key being pressed on the computer, a sudden significant change in sound level (such as a performer saying “bah”) or a sensor sending a message about location etc.. Each signal is sent to the system together with information about what sort of event it is. This additional information can be used or not as a situation demands.

Examples
The general form, which will trigger on *any* signal event is: -

(if) Dean gives a signal

To specify the type of input, use a type keyword in front of signal: -

(if) Dean gives a mouse signal

To specify additional things, add keywords (see below for which words can be used with each input type): -

(if) Dean gives a double mouse signal

mouse signal e.g. Dean gives a mouse signal	
This type of signal is generated in response to a mouse click. It has 3 possible additional properties, *short*, *long* and *double*.	
s*hort*	a ‘normal’ quick mouse click. e.g. Dean gives a short mouse signal
long	A click where the mouse button is held down for more than half a second before being released. e.g. Dean gives a long mouse signal
double	A fast double click e.g. Dean gives a double mouse signal

sound signal e.g. Claire gives a sound signal	
This type of signal is generated in response to a sudden sound pulse often through a performer’s headset. It has 3 possible additional properties, *short*, *long* and *double*.	
s*hort*	a short sound, for example “ki”.

	e.g. Claire gives a short sound signal
long	A sound in which a high level is maintained for more than half a second, for example “shar”. e.g. Claire gives a long sound signal
double	A double sound impulse, where the second sound starts within a third of a second of the first, for example, “di ka”. e.g. Claire gives a double sound signal

key signal e.g. ethan gives a key signal	
This type of signal is generated when the performer presses and releases a single key on their computer’s keyboard. It has 3 possible additional properties, *short* and *long* together with the letter value of the key being pressed.	
s*hort*	a ‘normal’ quick key press. e.g. ethan gives a short key signal
long	A key press where the key is held down for more than half a second before being released. e.g. ethan gives a long key signal
Letter value	The letter value of the key – for character keys, this is the character that would appear on screen if only that key were pressed, the letter is placed in speech marks. For control keys the following keyword can be used: - RETURN. SPACE is also allowed instead of “ “. e.g. ethan gives a “z” key signal ethan gives a RETURN key signal

Text events

Performance events involving text can, in principle, be of two types; typed or spoken (only the former is currently implemented). They are distinguished through the use of different keywords; *types* and *says*. Both types of event need the information to be matched to be specified either using a string of characters enclosed in speech marks or through a keyword. At the present time, both are synonymous in event descriptions.

Allowable keywords are; NOTHING and ANYTHING.

Examples

(if) susan types “welcome”

(if) ali types ANYTHING

If speech to text is implemented, the following can be used

(if) kate says “hello”

(if) mike says NOTHING

types e.g. Molly types “I wandered lonely as a cloud”
This type of event gives access to text input. The event is sent when the performer presses the RETURN key. Note that the performer client cannot support both typed text input and key and mouse signals at the same time.

MIDI

Performance events involving MIDI can be collected by the performer client in 2 different ways – as individual events or completed notes. These collection settings are set in the performer client.

Events generates a midi signal each time certain sorts of midi events are received by the client. They are described in event specifications by the keywords *gives a*, see below for details.

Notes generates a separate performance event for each note that is played (i.e. one event for each noteOn-noteOff pair) and are described in event specifications using the *plays* keyword.

The following 2 collection strategies may be implemented in future versions: - *Phrases* generates a a list of notes played whenever the player pauses for more than 2 seconds and no notes are being held.

Statements are similar to phrases but they are generated (as a performance event which includes a list of notes played) whenever the player presses a particular key (usually the top or bottom note on a keyboard).

MIDI note events are specified using the *plays* keyword together with a short piece of text identifying the pitch to be matched. Notes are specified by letter name (in upper case) and the modifiers ‘#’ for sharp and ‘b’ for flat. There should then be a space and a number indicating which octave is wanted. The whole note specification should be contained in speech marks. The keywords NOTHING and ANYTHING are allowed.

Examples

(if) sophie plays “A 4”

(if) ian plays “F# 2”

(if) frank plays ANYTHING

plays e.g. rashid plays “C 4”
This type of event gives access to high level midi input. When the event is sent depends upon the performer client settings. It will be when one of the following conditions is met: - Note - when a noteOn-noteOff pair is completed

Individual MIDI events are captured as signals

midi signal e.g. yolanda gives a midi signal	
This type of signal gives event-level access to midi input devices. It generates a lot of data and *midi input will generally be better gathered using higher level MIDI events* (see above). It has 2 possible additional properties, *eventType* and *note*.	
eventType	The type of MIDI event that generated the signal. Possible types are noteOn, noteOff, controlChange, programChange and pitchBend. e.g. yolanda gives a noteOn midi signal
note	A short piece of text which identifies the note (if any) that is associated with the MIDI event. Middle C is identified as “C 3”. Notes are specified by letter name (in upper case) and the modifiers ‘#’ for sharp and ‘b’ for flat. There should then be a space and a number indicating which octave is wanted. The whole note specification should be contained in speech marks. e.g yolanda gives a “G# 5” midi signal yolanda gives a “Bb 0” midi signal Note that the above examples will trigger on both the starts and ends of midi notes. To get a single notification for each note that is played, use the dedicated MIDI event type (see *play* above).

System events

The system is able to generate events arising from the *performance*, time based performance divisions called *sections*, representation of physical locations called *zones* and the state of *rules*.

performance

e.g. performance is active

Performances have 2 qualities that can be used in specifying events; activity and elapsed time.

When a performance is started, it is set to active and its clock starts counting the seconds since the start performance command was issued. To create an event which is fired at a particular point in a performance, test for the number of seconds: -

```
performance time is 120
```

or minutes: -

```
performance time is 2 minutes
```

'more than' and 'less than' are also allowed: -

```
performance time > 45
performance time is more than 5 minutes

performance time < 360
performance time is less than 6 minutes
```

zone

e.g. zone zone1 is populated

Zones are ways that physical locations can be dynamically represented by the system. They have 2 qualities; whether they are populated or empty and a 3 dimensional position.

Zones are identified by a name:-

```
zone myzone
```

You can use sense data to update their properties (see commands section) and then make rules which fire when a zone is 'in' a particular location:-

```
zone myzone position is (1.5,2.0,0.0)
```

or when the zone is populated: -

zone myzone is populated

or when one of the zone's position co-ordinates is equal to, less than or more than a specific value: -

zone myzone x is 5.0
zone myzone y is less than 1.3
zone myzone z > 10.5

section

e.g. section interlude1 is active

Sections are ways that time-based divisions of the performance can be dynamically represented by the system. They have 2 qualities; whether they are active or inactive and an elapsed time.

Sections are identified by a name:-

section intro

Sections are created when a new section name is encountered by the system, but they are not started (set to active and their clock reset to 0 and started) unless the start command is used: -

start section intro

Sections are stopped by the stop command: -

stop section intro

When a section is started, its activity can be used to fire events in rules: -

(if) section intro is active

Its elapsed time, counted in seconds, can also be used in rules (until a section is started, it remains at 0):-

(if) section intro time > 4 minutes
(if) section main time is less than 20

Commands

Commands cause the system to do something. They can be issued by the user or triggered by a rule if its triggering circumstances arise. All commands require a target reference (e.g. performer 2, rule 7), although this is implicit in a small number of cases (see entry for 'get' below) and a data value. The particular action triggered by a command is often dependent on both the keyword used and the data type supplied.

The ordering of elements in a command usually follows one of 2 syntactical models, as far as possible following that of 'natural' English usage for each keyword. In all cases, the command keyword must be the first item.

Examples

set zone upstage to populated (keyword-target-data)

say "hello" to kevin (keyword-data-target)

Commands can be divided into 2 main categories based on the type of action they cause to happen; *cuing* and *data management* commands.

Cuing commands

These cause messages or orders to be sent to performer clients, usually accompanying one or more pieces of data and requesting the client to display it to the performer in an appropriate manner.

Examples

say "hello" to daniel (speaks)

play "D# 4" to daniel (plays a tone)

show "start section 7" to daniel (displays text)

show image score1.gif to daniel (displays image)

Data management commands

These direct the system to carry out actions on its own internal data representation. This could cause a change in the system operation, for example starting and stopping performances, or they might alter the database of stored rules.

Examples

start section intro

set rule 2 to INACTIVE

set zone camera1 position to (1.2,2.0,0.0)

Identifying targets

Performers and rules are identified by name. If the first word of a target is not a keyword, it is assumed to be the name of a performer or rule.

The EVERYONE keyword is also allowed for cuing commands (those whose target is a 'performer').

Examples

say "hello" to joseph

show image image1.jpg to EVERYONE

Cuing Commands

say e.g. say "hello" to michelle	say *data* to *target*
The say command sends a text string to a performer client and asks it to speak it to the performer. The text to be spoken should be enclosed by speech marks. The character sequences .txt and .html are not allowed in text strings as they are used to distinguish strings from filepaths (future performance clients will offer the capability to read from text files). If the performer client does not have speech capability, the text will be displayed on screen instead.	

<table>
<tr><td colspan="2">show

e.g. show “hello” to monica</td><td>show data to target</td></tr>
<tr><td colspan="3">The show command sends a text string or an image to a performer client and asks it to display it on the computer’s monitor.

Note that the current version of the performer client cannot display both text and images at the same time. However other modes of feedback (speech, MIDI etc.) do not affect the visual display.

When a performer client displays a new text or image item, it displaces whatever was previously being displayed.

Displaying text from a file is not yet implemented.</td></tr>
<tr><td>text</td><td colspan="2">For text, the data expression may be either an absolute expression or a file reference (only the former is currently implemented). For absolute expressions, speech marks should enclose the text.

e.g. show “start playing now” to Jane

show polemic.txt to Tony

The character sequences .txt and .html are not allowed in absolute expressions as they are used to distinguish strings from filepaths. File names may not include spaces.

For files, the text should be formatted as plain text or as (simple) html.</td></tr>
<tr><td>images</td><td colspan="2">For images, only a file reference is allowed which must be preceded by the keyword image. File names may not include spaces. Images should be stored in jpeg or gif format.

If the image is larger than the display area (950 wide, 600 or 485 high depending on the keyboard input mode) will be scaled to the display area’s size. The proportions of the image are preserved

e.g. show image xmas_card.jpg to EVERYONE

The image will first be downloaded to the performer client’s computer – which may cause a delay if files are large and networks are slow. However, the client caches all downloaded assets locally so that they are available instantly for subsequent use.</td></tr>
</table>

<table>
<tr><td>play

e.g. play “C# 6” to Django</td><td>play data to target</td></tr>
<tr><td colspan="2">The play command sends a text string identifying a MIDI note or a file reference to a performer client and asks the client to play the file through the appropriate playback system. The file may be either a MIDI file or a sound file – the type is indicated using the midi or sound keywords. File playback is not yet implemented.

Note that playing back a sound file may interfere with spoken feedback.

When a performer client receives a request to play a new piece of sound or MIDI information, it interrupts any playback that is already taking place.

There is currently no way to set instruments for MIDI playback using rules, although this can be done manually in the performance client.

Playback is currently restricted to a single MIDI channel – which is selectable in the performance client.

Only MIDI note playback is currently implemented.</td></tr>
<tr><td>MIDI note</td><td>Notes are specified by letter name (in upper case) and the optional modifiers ‘#’ for sharp and ‘b’ for flat. There should then be a space and a number indicating which octave is wanted. The whole note specification should be contained in speech marks.

e.g. play “Bb 4” to Jane</td></tr>
<tr><td>MIDI file</td><td>For MIDI files, a file reference, must be preceded by the keyword midi. File names may not include spaces.

e.g. play midi fanfare1.mid to performer 5

e.g. play midi 10.0.1.1/midi/fragment6.mid to performer 5</td></tr>
<tr><td>sound file</td><td>For sound files, a file reference, must be preceded by the keyword sound. File names may not include spaces.

Permissible sound file formats are .aif, .wav and .mp3

e.g. play sound stretched.aif to performer 5</td></tr>
</table>

Data and system management commands

get e.g. get me rule SysName_RULE_14	get *me* rule *rule_name* get *me* rule-list
The get command causes a data object in the system to be retrieved and returned to the target (which has to be *me*, the requesting client at present)	
rule	For rules, the events and commands making up a rule are returned in an XML format. At present, a rule's state and labels are not returned.
rule-list	Returns an XML listing of all current rule's (unique) system names. These are in the form of SysName_RULE_n, where n is an ID number

set e.g. set rule 2 to INACTIVE set zone origin position to (2.5, 2.5, 5.0)	set *target* to *data*
The set command causes a data object in the system to be loaded with a new value. There are a number of allowable targets each of which affects the permissible values for *data*; performance, rule, section (not implemented).	
rule	For rules, data can have two values, ACTIVE and INACTIVE. If a rule is active, its conditions are considered by the system when it judges if any significant events have taken place. e.g. set rule 2 to ACTIVE A rule can make itself inactive as one of its commands – meaning that it will only be triggered once.
section	For sections, data must be a text string. e.g. set section "transition 2" to active Assigning section a new name (even to the same name) causes the system to reset the section time. Section can thus be used in event definitions to make commands conditional on stages of a performance. Progress through a section is then given by section time. e.g. (if) section time > 1 minute
performance	A performance can be either ACTIVE or INACTIVE. When the system starts, all performances are set to inactive with a

	performance time of 0. Setting performance to active can also be done using the *start* command (see below). e.g. set performance to active If a performance is active, its performance time will be continually increased. When it is stopped (by setting it to inactive), the performance time is reset to zero.
zone	A zone can be set to both a position and to 'populated' or 'empty'. Zones are referred to by name. e.g. set zone upstage to populated set zone avater1 position to (0.0, 3.1) If the zone name has not been previously encountered by the system in the project you are working in, it is created. Positions can be 2 dimensional or 3 dimensional and are specified 2 or 3 numbers within parentheses, separated by commas:- e.g. (3.2,1.0,-2.3)

start e.g. start section introduction	start performance start section *name*
The start command is a synonym for set *formalUnit* to active (see *set* section above) It can be used with performance or section, when a name for the section ust be supplied. When the command is executed, the performance or section clock is started and is updated until the section or performance is stopped, reset or set to inactive.	
performance	For the performance, no further information is needed. e.g. start performance The performance is always set to stopped (or inactive) when it is created or loaded from the database.
section	For sections, if the section has already been involved in the performance as either an event source or as the target of a command, it is set to active and its internal elapsed time clock is

	started. If it has not been involved in the performance, it is created and its clock started. e.g. start section introduction Sections are not saved by the system – they are created as needed during a performance (this process should be transparent to the user – if you mention a section and it does not already exist, it will be created)

stop e.g. stop section introduction	stop performance stop section *name*
The stop command is a synonym for set *formalUnit* to inactive (see *set* section above) It can be used with performance or section, when a name for the section must be supplied. When the command is executed, the performance or section clock is stopped and set to 0.0.	
performance	For the performance, no further information is needed. e.g. stop performance The performance is always set to stopped (or inactive) when it is created or loaded from the database.
section	For sections, if the section has already been involved in the performance as either an event source or as the target of a command, it is set to inactive and its internal elapsed time clock is stopped and set to 0.0. If it has not been involved in the performance, it is created. e.g. stop section introduction Sections are not deleted by stop so that their activity may still be used in event descriptions e.g. (if) section introduction is inactive However, note that there is currently no way to test if a section has ever been active.

Keyword listing

ACTIVE
ANYONE
ANYTHING
controlChange
delete
double
event
EVERYONE
get
give a
gives a
image
INACTIVE
is
is more than
is less than
key
long
make
midi
minutes
mouse
movie
NOONE
noteOff
noteOn
NOTHING
osc
Performer
performance
performance time
pitchBend
plays
position
programChange
RETURN
rule
rule-list
says
seconds
section
section time
short
show
signal
sound
SPACE
time

to
trigger
types
x
x-position
y
y-position
z
z-position