# The Future of the Clockwork Programming Language

Martin Leadbeater

August 26, 2013

**Abstract**

This paper describes various language features that will be added to Clockwork in the future

## 1 Introduction

- Floating point calculations
- Sets and relationships
- Lists
- Streams (also generators)
- Replace the current symbol table and Value system with a more efficient and robust implementation
- Type conversions (x AS type)
- Substates
- Exceptions
- Conditions
- Delayed messaging (sending messages after a delay and cancelling..)
- Wall time (long period timers)
- Support of parallel execution
- Type extensions
- External function calls
- Passing SELF to an internal object
- Sharing properties or data
- Dynamic allocation (ie creating and destroying data items)
- Dynamic configuration (creating and destroying machines)

Since machines are defined statically there is no way to create a new machine or destroy one that is not needed. The language uses properties to store values and although machines can be named, there is no way to make a property that refers to a machine. It is not even possible to save the state of a machine and restore that state later.

## 2 Feature discussion

Clockwork is currently based around the notion of machines, states and properties. Machines cannot carry data per-se but since the property system provides for simple key-value objects, the effect can be simulated at a basic level. Machines also need the notion of a working-set of data that is being processed or held. Traditionally data is grouped into 'structure's or 'object's and these terms and their common usage seems fine. More generally, a 'working set', if we chose to use the term, would refer to a collection of various values, structures and objects.

### 2.1 Value system

The way property values are stored in machines desparately needs an overhaul; the current system is build on a C type structure and uses runtime type comparions. A new version will use C++ virtual methods that bind functions at compile time.

## 2.2 Floating point calculations

## 2.3 Sets and relationships

We need to decide whether bits that are allocated to symbols should be allocated in least-significant or most-significant order. Perhaps a mapping function should be used (like a C union), with this in mind, defining three symbols, x,y and z to a block of bits would map bit 2 to x, bit 1 to y and bit 0 (least significant) to z. This gives a nice mapping for those who visualise the bits in a block and want to map the symbols over the block.

```
cutter1 Cutter;
cutter2 Cutter;
B BITSET [a,b,c,d]; # defines symbols a..d for bits 0..3
C BITSET [r:8,g:8,b:8]; # defines symbols r,g and b for bits 0..23 of C
INCLUDE a,d IN B
x := B AS INTEGER;
cutters SET[cutter1, cutter2];
s INCLUDES? [a,b,c];
```

Sets could be implemented as a machine where each combination of members is a distinct state there doesn't seem to be much value in this. It may be useful to use three states: empty, nonempty and full.

### 2.3.1 Enumeration

There should be a way to trigger events for each item in a set:

```
EACH item IN set DO command
```

### 2.3.2 Events

Sets should send messages on state change between empty, nonempty and full. It may be useful to send events as each item enters/leaves but at this point that seems excessive and of little value.

## 2.4 Streams

A Stream is an ordered collection of data that can be grouped into fixed size units (bits, bytes, chars etc); it has a current value and position. the position can be moved forward or backward and the value can be read and changed. Streams may have a beginning and end and has a state of closed, valid or invalid. In the closed and invalid state, the current value cannot be read; if the stream is moved past the end or before its beginning, it becomes invalid. A Reader is a special case of a Stream in which the data cannot be changed, similarly, a Writer is a Stream that can only be written to.

```
s READER "example.dat":1; # read one bit at a time
done WHEN s IS invalid; # terminating condition
start WHEN s IS closed;
one WHEN s.curr == 1;
zero WHEN s.curr == 0;
ENTER start { SEND open TO s }
ENTER one { LOG "1"; SEND forward TO s }
ENTER zero { LOG "0"; SEND forward TO s }
```

## 2.5 Lists

Procedural or functional style? A functional language tends to use functions such as car and cdr. Similar to sets, it should be possible to enumerate objects and also to receive events as items are added or removed from the list.

## 2.6 Substates

## 2.7 Exceptions

## 2.8 Conditions

## 2.9 Support of parallel execution

The most direct method to support parallel execution is to identify separate chains of dependency within the machine graph. Thus, for example, changes that occur in one input may be able to be completely processed in parallel with another chain if the steps are independent of changes that occur in another input. [yuk reword]

More explanation required

## 2.10 Type Extensions

Type extensions provide for adding parameters, statements to the handlers within a state machine, fields and substates.

```
A MACHINE { ... };
B MACHINE EXTENDS A <additional parameters> {

    <event handler extensions>
    <additional fields>
    <state extensions>

}
```

More thought is required here, what I am planning is that clauses within B can be defined as separate or as extensions of those in A, for example INIT { ... } simply defines a handler for the INIT state in B. Whereas INIT EXTENDS A.INIT { ... } defines extra steps that are added to the handler already defined in A.