

DESL: A Generic, Lisp-Based Discrete-Event Simulator

Roy M. Turner

Technical Report 2010-3
Department of Computer Science
University of Maine
Orono ME 04469

Contents

1	Introduction	2
2	Events	2
3	Time	3
4	Event Queues	4
5	Installing DESL	4
6	Loading DESL	5
7	Running DESL	5
7.1	Creating a DESL instance	5
7.2	Dealing with events	6
8	Detailed Documentation	6
8.1	The make-desl macro	6
8.2	The DESL Class	7
8.3	Event Classes	9
8.4	Time-related methods and functions	18
8.5	Miscellaneous Functions	23
	Index	25

1 Introduction

DESL (Discrete-Event Simulator/Lisp) is a generic discrete-event simulator implemented in Lisp for use by projects at MaineSAIL,¹ in particular MEME ((Maine Ecological Model of Estuaries) and CoDA (Cooperative Distributed AOSN controller). Although there are many software packages to support discrete-event simulation (DES), there are few that are suitable for use with Lisp-based software without interfacing to foreign functions.

DESL is object-oriented (using CLOS) and is meant to support agent-based modeling (ABM)² [Grimm & Railsback, 2005]. In designing DESL, we have tried to make it as general-purpose as possible. For example, there are several ways to specify events, the most general of which simply tells DESL the time of the event and a program-specific description of the event.

Basically, someone wishing to use DESL in their program would load it and instantiate a DESL object (of class `DESL`). New events would be sent to DESL, and its methods would be used to process events, either by notifying the caller or by taking the actions they specify (see below). The simulated time is also maintained by DESL and updated as events transpire, although of course the user's program can maintain its own copy of the current time.

In the rest of this document, we first discuss DESL's event model, how it handles time, and the various kinds of event queues DESL supports. We then describe how to install (if necessary), load, and run DESL. What follows that is more detailed documentation of the code itself.³

2 Events

Events are the basic transaction for any discrete-event simulator, including DESL. The user or user program adds events to DESL, which stores them in an *event queue*. Events are processed by asking DESL to run, either via the `run-one` method (runs until a single event has been processed), `run-n` (runs until n events have been processed), `run-all` (run all events) or `run-until` (runs until all events occurring up until some time have been run). As events are run, the current simulation time is updated by DESL.

An event, at its simplest, is just an arbitrary Lisp expression that describes the event, and a time at which that event is to occur. For example, in a discrete-event simulator used for predator-prey modeling, an event might be something like: at time t , a prey individual lays a clutch of eggs. This might be represented as:

```
t (LAY-EGGMASS PREY05)
```

This event would be added to the simulator using `add-event`:

```
(add-event desl-instance t :description (lay-eggmass prey05))
```

For very simple events like this, DESL's `run-one` method (e.g.) would return the time and description of the event to the caller. It would then be up to the user program to do something with that information. Internally, simple events like this are represented by objects of class `DESL-event`, which is based on the `event` class.

DESL provides a richer set of event types, however, also represented as classes built on the `event` class. One of these is a *callback event*, represented by the class `callback-event`. Here, the user specifies a callback function, and when the event occurs, DESL calls that function, giving as arguments the time of the event and the event's description.

An `OOP-callback-event` is a variant of this. Here, DESL calls a method of an object rather than a callback function. (OOP here means object-oriented programming.) Both the callback method and the object are specified by the user, and the callback method is called with the object, the time, and the description as arguments.

An `autonomous-event` is one that handles itself when it occurs. This event has a handler method associated with it (which can be specified by the user), and this handler is called with the event itself as the

¹<http://MaineSAIL.umcs.maine.edu>

²Also called individual-based modeling (IBM) by some authors in the computational ecology literature.

³This file was produced using MaineSAIL's LP/Lisp literate programming tool [Turner, 2010].

argument (i.e., the method of the event object is called) when the event occurs. The intent here is for the user to create his or her own event classes based on `autonomous-event` that do application-specific things when they occur.

These event types can be instantiated by the user directly and added to DESL via its `add-event` method. However, convenience methods exist for doing this as well: `add-callback-event`, `add-object-callback-event`, and `add-autonomous-event`. In addition, `add-event` can determine, based on its arguments, which kind of event you mean. See the documentation for these methods for details.

You can see what the next event is at any time using `peek`, and you can get the next event, without processing it, using `next-event`. In addition, you can print all events using `print-events`. See the documentation for these methods for more details.

Events can be deleted from the queue using the `delete-event` method of DESL. This method can be told which event to delete by specifying the event object itself (of class `event` or its subclasses). Alternatively, it can find events based on the time they occur, by their description, or both.

Events are deleted either by just marking them as such, then ignoring them when they occur, or by actually removing them from the queue. This is user-selectable. The former is likely quicker, since the queue won't have to be reorganized, but the latter saves the space they would have taken up, and, in the case of many deleted events for some queue types, may make the queue as a whole less efficient.

3 Time

DESL's basic time unit is the second, and all of the internal times are represented as such. However, DESL allows the user to use whatever time unit makes sense for the application. Supported units include milliseconds, seconds, minutes, hours, days, weeks, and years. (Months were skipped, since they have variable numbers of days.) This is specified when a DESL object is instantiated using the `:time-units` keyword argument. Appropriate translations occur based on this setting when you add an event, when an event occurs, or when time is printed.

By default, a DESL instance is created using the current system time (the so-called *universal time*) as both the simulation start time (instance variable `start-time`) as well as DESL's current time (`current-time`). This is the number of seconds since the beginning of 1/1/1900.

This will not be, for most users, the time desired, and it can be overridden by using the `:start-time` argument when calling `make-instance`. However, this is not recommended unless you really know what you are doing. Instead, call the `set-start-time` method, which allows the time to be specified in a much friendlier format.

Time internally is similar to a universal time, except that it is specified as a double float rather than an integer. This is so that DESL can support milliseconds directly.

Several time-related functions and methods are available, including:

- `M->S`: Convert minutes to seconds.
- `H->S`: Convert hours to seconds.
- `D->S`: Convert days to seconds.
- `W->S`: Convert weeks to seconds.
- `MO->S`: Convert months to seconds; the month length is settable, and defaults to 30.
- `Y->S`: Convert years to seconds. This assumes that years are 365 days, and it ignores anything like leap years.
- `current-time`: This returns the current time, in seconds. **Note:** This is not a universal time, but rather a double float version of a universal time. Consequently, it cannot be passed directly to something like `decode-universal-time`, which expects an integer (fixnum). Instead, you should do a `floor` first, keeping the remainder (the second return argument), which is the fractional seconds.
- `current-timestring`: This returns the current time as a string. There are three keyword arguments this accepts:
 - `:absolute` – format the actual current time value stored in `current-time`
 - `:relative` – format the current time minus the start time

- `:as-units` – if `t` (the default), then format the time as the number of those units it represents; otherwise, a format is used that splits the time into years, weeks, months, etc., starting with the first non-zero quantity

If neither `absolute` nor `relative` is set, then the value of the DESL instance variable `time-is-absolute` is used.

- `now-plus-interval`: Returns a new time that is the current time plus whatever interval you specify, based on the keyword arguments `:seconds`, `:minutes`, etc.
- `simtime-to-string`: Convert a simulation time to a string; this does what `current-timestring` does (in fact, it's what `current-timestring` calls), and has an additional keyword argument, `:time`, which defaults to the current time.
- `time-to-units`: This converts a time (specified as a number of seconds) to other units, based on the value of its `:units` keyword argument, which takes the values `:seconds`, `:minutes`, etc.
- `parse-time`: This will parse a simulation time (in seconds) into multiple values (i.e., use `mbind` (a short form of `multiple-value-bind`) or `msetq` (a short form of `multiple-value-setq`) to catch them): milliseconds, seconds, minutes, hours, days, weeks, and years, in that order.

See the documentation for these functions for more details.

4 Event Queues

DESL supports several different queue types. At the current time, its queue can be either a sorted list, a heap, or calendar queue. A *calendar queue* [Brown, 1988] is a data structure optimized for managing time-related information. This is the default queue type, since it is faster than the other two for discrete-event simulation applications.

The type of queue you wish to use is specified when instantiating DESL, using the `queue-type` keyword argument to `make-instance`. Giving this one of the values `:calendar-queue`, `:calendar`, or `:cq` selects the calendar queue type, while a priority queue (implemented as a list) is selected with `:priority-queue`, `:priority`, or `:pq`, and a heap is selected with `:heap`.

Note that heaps and calendar queues perform approximately the same, with calendar queues being a bit faster;⁴ Priority queues, which are implemented (as of this writing) as a standard doubly-linked list, perform much, much worse.

5 Installing DESL

If you are a MaineSAIL user using one of our machines, you can skip this section.

If you are not at MaineSAIL, there are several things to do to install DESL. First, you need to install a copy of the MaineSAIL utilities, which you should have received when you received the DESL files. Once those are installed, then DESL's defsystem file, `desl.system`, will need to be set up correctly for your location. There are some lines at the top of the file that need to be changed. In the line:

```
(defparameter *desl-basedir* "/home/cdps/DESL")
```

replace `/home/cdps/DESL` with the name of the directory in which DESL is installed. The two lines

```
#-:cdps-translations
(load "/home/cdps/cdps.translations" :verbose nil)
```

are meant to load a file containing rules to translate logical pathnames. You will need to either replace that filename with one pointing to your own translations file, or replace those lines with:

```
(setf (logical-pathname-translations "desl")
      '((";"fasl;*.*)" ,(concatenate 'string *desl-basedir* "/FASL/"))
      ("**;*.*)" ,(concatenate 'string *desl-basedir* "/"))))
```

⁴When output verbosity is low; with verbosity set higher, then performance of the heap and calendar queue versions is about the same.

If you wish to allow DESL to be loaded via `require`, then you will have to tell Lisp where it is. This can be done by creating a directory in which you will store a file telling Lisp where to find DESL, then telling Lisp where to find *that* directory. (This is all for Allegro Common Lisp; something similar is likely to be possible for other Lisps as well.) Somewhere in your system-wide `siteinit.cl` or your own `.clinit.cl` file, add the lines:

```
(setf (logical-pathname-translations "mysite")
      '((";**/*.*" "your-directory")))

(setq sys:*require-search-list*
      (append sys:*require-search-list*
              (list "mysite:;.fasl" "mysite:;.lisp")))
```

Now put a file, `desl.lisp`, in that directory that contains the lines:

```
(require :utilities)
(load "desl:;load")

(provide :desl)
```

and compile it.

Note that in order for DESL to work correctly, the utilities need to be installed so that they can be loaded via `require` as well.

Once this is done, you should be able to load and run DESL as described below. Prior to running DESL for the first time, however, you should compile it. Enter Lisp and load the defsystem file, `desl.system`. They do:

```
(compile-system :desl)
```

You should now be all set.

6 Loading DESL

DESL is defined as a Lisp system. If your installation is set up for using `require` (see above), then can be loaded by:

```
(require :desl)
```

This works for users at MaineSAIL.

If not, you will need to manually load the defsystem file, `desl.system`, then do:

```
(load-system :desl)
```

The defsystem file is found in the installation directory.

7 Running DESL

7.1 Creating a DESL instance

In order to use DESL in your application, you first must create an instance of the class `DESL`. This is done using either `make-instance` or the `make-desl` macro. For `make-instance`:

```
(make-instance 'desl keywords)
```

Valid keyword arguments include:

- `:time-units` – specify which time unit is in use; by default, it is seconds.
- `:queue-type` – which type of queue you want; options are calendar queue (specified via values `:calendar-queue`, `:calendar`, or `:cq`), priority queue (`:priority-queue`, `:priority`, or `:pq`), or a heap (`:heap`).

After making the instance, you should set the start time, unless you are happy with it being the current (universal) time. This is best done with the `set-start-time` method. If this is called with its `now` keyword argument set to `t`, or if none of the other keyword arguments are set, the current time is used. If `zero` is set to `t`, then the start time is 0 seconds (which is likely to be the most useful). If `raw` is set to a number, then that number, considered to be a kind of universal time, is used. Otherwise, its other keywords, corresponding to seconds, etc., are used to set the time to a universal time (i.e., a time offset from 1/1/1900). These are: `second`, `minute`, `hour`, `day`, `month`, and `year`.⁵ For example,

```
(set-start-time DESL-instance :year 2009 :month 3 :day 24 :hour 11 :minute 0
                :second 0)
```

would set the start time to 3/24/2010 11:00:00.

You don't have to set every one of these. The method tries to do something semi-intelligent to fill in the missing values. If you specify a more specific value (e.g., `minute`) and leave the more general values unset, then they default to the values for the current time. For example, if you specify:

```
(set-start-time DESL-instance :day 24 :hour 11)
```

on 3/12/2010, then the time would be set to 3/24/2010 11:00:00.

NOTE: Calling this creates a new event queue – so it would be unwise in general to call it except before DESL begins work.

A more convenient way of creating a DESL instance is provided by the `make-desl` macro, which combines the functionality of the `make-instance` call and the call to `set-start-time`. It takes all of the keyword arguments you would normally give `make-instance` for the DESL class (see the documentation of that class), as well as another keyword argument, `start-time`. This is a list that specifies the start time in the same form as the arguments to `set-start-time`. For example,

```
(make-desl :queue-type :cq :start-time (:day 24 :hour 11))
```

would create a new DESL object with an event queue implemented as a calendar queue and a start time as described above. (Note that since `make-desl` is a macro, you don't have to quote the `start-time` value.)

7.2 Dealing with events

Adding events to the queue is done with `add-event`, `add-callback-event`, `add-object-callback-event`, or `add-autonomous-event`. The documentation for these methods is with the methods, below. (See the index.)

To process the next event, use `run-one`, `run-n`, `run-all`, or `run-until`. See those methods' documentation for how to call them.

Events can be deleted with `delete-event`. This can be given an event object to delete (as the value of the `event` keyword argument), or it can find events to delete based on the event description (specified as the value of the `description` keyword) or a time description (the value of `time`). For a description of how to specify `time` and how to specify how `description` is handled, see the documentation for `find-event-by-time` and `find-event-by-description`.

At the current time, event deletion is only implemented for calendar queues, although the base types of priority queues and heaps do support event removal. Also, there is no mechanism yet for having DESL run as an autonomous process to which messages about events can be sent and from which they can be received.

8 Detailed Documentation

8.1 The `make-desl` macro

Macro `make-desl`

⁵At the moment, milliseconds as part of the start time are not supported.

The `make-desl` macro is meant to simplify the instantiation of a DESL instance for the user. It takes basically all of the keyword arguments valid for `make-instance` when creating DESL and passes them to `make-instance`. (See the class definition's documentation to see what these are, as well as what their defaults are.) It also takes a `start-time` keyword argument that specifies how the time should initially be set. If this is `nil`, then `set-start-time` is not called. Otherwise, the description of the start time is taken apart and passed to `set-start-time`.

Example time descriptions include:

```
(make-desl :start-time (:now t))
(make-desl :start-time (:zero t))
(make-desl :start-time (:hour 20 :minute 0 :second 3))
```

Note that you do not have quote the time description; since `make-desl` is a macro, the values won't be evaluated until appropriate.

```
[00001] (defmacro make-desl (&rest args)
[00002]   (let (stime actual-args)
[00003]     (setq actual-args
[00004]       (loop while args
[00005]         when (eql :start-time (car args))
[00006]         do (setq stime (cadr args))
[00007]           (pop args)
[00008]           (pop args)
[00009]         else
[00010]         append (list (pop args) (pop args))))
[00011]     '(let ((inst (make-instance 'desl ,@actual-args)))
[00012]       (set-start-time inst ,@stime)
[00013]       inst)))
[00014]
```

8.2 The DESL Class

To create a new DESL engine, the application program creates a new instance of the class `DESL`. The skeleton of this class is described in this section. There are many instance variables, which can be grouped into several classes. We will discuss each class and list those variables below.

In order to support multiple programs, `DESL` needs to be able to handle a variety of time scales and extents. For example, `MEME` needs to have time measured in minutes or hours, or even days, while `CoDA` needs a granularity of seconds. On the other hand, `CoDA` so far has only needed temporal extents of an hour or so, while `MEME` will need to be run for a simulated month or more.

The instance variables for this class hold all the time- and event-related information that `DESL` needs. Time-related instance variables include `base-time`, `current-time`, and `time-units`. The latter is self-explanatory, and possible values are the ones expected: `:milliseconds`, `:second`, `:minute`, `:hour`, `:day`, `:month`, or `:year`.⁶

The start time of the simulation is contained in `start-time`, which is similar to a universal time, in that it is some number of seconds, but it is different in that it is a double float, and it may not be in reference to some universal starting point, unless it is actually converted from the true universal time and is treated as such. It defaults to the current time. This instance variable should not be set directly unless you really know what you're doing; rather, use the `set-start-time` method. The current simulation time is contained in `current-time`, which is also a universal time. This is set by the events as they are processed.

The primary event-related instance variable is `event-queue`, which holds the queue of events. By default, this is a calendar queue (`calendar-queue`). You can also specify, using the `queue-type` keyword/instance variable, other queue types. At this time, we support using priority queues (of class `priority-queue`, and specified with `:priority-queue`, `:priority`, or `:pq`) or heaps (of class `heap-pq`, specified using `:heap`).

⁶We do not anticipate simulating things that need sub-millisecond timing; this could be added in the future, though.

Note that most of what gets done by DESL is done by the events themselves as they are processed, either directly or via callbacks to the program that generated the event.

DESL has two ways to delete events.⁷ The `delete-event` method can either mark an event as deleted, but leave it in the queue, or it can actually remove it from the queue. The former is faster, but the latter saves space. The choice is up to the user. The `ignore-deleted-events` slot determines whether events that have been marked as deleted are processed or not. If the value is `t`, then deleted events aren't processed, and their presence is invisible to the user. Otherwise, they are processed as if they aren't deleted.

```
[00015] (defclass DESL ())
[00016]   (
[00017]     ;; time-related:
[00018]     (valid-time-units
[00019]       :initform
[00020]       '(:milliseconds :seconds :minutes :hours :days :weeks :years))
[00021]     (time-units :initarg :time-units
[00022]       :initform :seconds)
[00023]     (time-is-absolute :initarg :time-is-absolute
[00024]       :initform nil)
[00025]     (start-time :initform (get-universal-time))
[00026]     (current-time :initform nil)
[00027]     ;; event-related:
[00028]     (valid-queue-types
[00029]       :initform
[00030]       '(
[00031]         ((:calendar-queue :calendar :cq)
[00032]          (calendar-queue
[00033]            :start-time ?start-time
[00034]            :current-time ?current-time
[00035]            :resizing t
[00036]            :ignore-deleted-events t))
[00037]         ((:priority-queue :priority :pq)
[00038]          (PQueue
[00039]            :priority-eval-fcn #'event-time
[00040]            ))
[00041]         ((:heap)
[00042]          (heap-pq
[00043]            :value-accessor #'event-time
[00044]            ))
[00045]         ))
[00046]     (queue-type :initarg :queue-type :initform :calendar-queue)
[00047]     (event-queue)
[00048]     ;; bookkeeping info:
[00049]     (events-in :initform 0)
[00050]     (events-out :initform 0)
[00051]     (deleted-events :initform (make-hash-table :test #'equal))
```

The following can be set to `nil` if you don't need to delete events; that might speed up processing slightly.

```
[00052]   (ignore-deleted-events :initarg :ignore-deleted-events
[00053]     :initform t)
[00054]   (*message-handler* :initarg :*message-handler*
[00055]     :initform (make-instance 'message-handler))
[00056]   )
[00057] )
[00058]
[00059] (defmethod initialize-instance :after ((instance desl) &rest other-stuff)
```

⁷This has so far only been implemented and tested when using a calendar queue as the event queue.

```

[00060] (declare (ignore other-stuff))
[00061] (initialize-queue instance)
[00062]
[00063] (defmethod initialize-queue ((instance desl))
[00064]   (with-slots (current-time start-time valid-queue-types queue-type
[00065]               valid-time-units time-units) instance
[00066]     (let ((desc (cadr (assoc queue-type valid-queue-types
[00067]                           :test #'(lambda (a b)
[00068]                                     (member a b))))))
[00069]       (unless current-time
[00070]         (setq current-time start-time))
[00071]
[00072]       (unless (member time-units valid-time-units)
[00073]         (error "Invalid time unit ~s specified for DESL." time-units))
[00074]
[00075]       (unless desc
[00076]         (error "Invalid queue type ~s specified for DESL." queue-type))
[00077]
[00078]       (make-event-queue instance desc)
[00079]     )
[00080]   )
[00081] )
[00082]
[00083] (defmethod make-event-queue ((self desl) description)
[00084]   (with-slots (event-queue) self
[00085]     (setq event-queue
[00086]       (eval
[00087]         '(make-instance ',(car description)
[00088]           ,@(loop with pairs = (copy-list (cdr description))
[00089]                 while pairs
[00090]                 append (list (pop pairs)
[00091]                               (let ((val (pop pairs)))
[00092]                                 (cond
[00093]                                   ((var? val)
[00094]                                    (slot-value self (varname val)))
[00095]                                   (t
[00096]                                    (eval val))))))))))
[00097]

```

8.3 Event Classes

Class `DESL-event`

This class is used when DESL creates an event for the user – that is, when `add-event` is called with just a time and description, not a user-defined event.

```

[00098] (defclass DESL-event (event)
[00099]   (
[00100]     (creator :initarg :creator :initarg :creator :initform nil)
[00101]   )
[00102] )
[00103]

```

Class `callback-event`

This class is used for those programs that want to be called when events occur. When an event of this class is processed, it calls `callback` with the event's time and description as its arguments.

```

[00104] (defclass callback-event (event)
[00105]   ((creator :initarg :creator :initform nil)
[00106]    (callback :initarg :callback
[00107]             :initform #'(lambda (time description)
[00108]                           (warn "Unhandled callback event ~s called at ~s."
[00109]                                 description time))))))
[00110]

```

Class OOP-callback-event

This is a subtype of the callback event for object-oriented programs. The `callback` function is assumed to be a method of the object stored in `object`; in this case, the method is called with `object` as its first argument, and with the event's time and description as its remaining arguments.

```

[00111] (defclass OOP-callback-event (event)
[00112]   ((object :initarg :object :initform nil)
[00113]    (creator :initarg :creator :initform nil)
[00114]    (callback
[00115]     :initarg :callback
[00116]     :initform
[00117]     #'(lambda (object time description)
[00118]         (warn "Unhandled callback event for object ~s encountered at ~s (desc=~s)."
[00119]               object time description))))))
[00120]

```

Class autonomous-event

This event class implements events that handle themselves. When this event type is encountered by DESL, it calls the event object's own event handler, which is named in `handler`. Users would subclass this event type and define their own event handler.

```

[00121] (defclass autonomous-event (event)
[00122]   ((handler :initform #'handle :initarg :handler)
[00123]    (creator :initarg :creator :initform
[00124]            nil)))
[00125]
[00126] (defmethod handle ((self autonomous-event))
[00127]   (with-slots (time description) self
[00128]     (warn "Autonomous event ~s called, but no handler; time=~s, description=~s"
[00129]           self time description)))
[00130]
[00131]

```

Method `add-event(DESL,event)`

Method `add-event(DESL,time,description,creator,from-now?)`

Method `add-callback-event(DESL,time,callback,creator,description,from-now?,from-start?,time-units)`

Method `add-object-callback-event(DESL,time,callback,object,creator,description,from-now?,from-start?,time-units)`

Method `add-autonomous-event(DESL,time,description,handler,class,creator,from-now?,from-start?,time-units)`

These methods are used to add events to the queue. There are two versions, one for when the user program creates an event object and adds it itself, and one for when it just wants to send in the time and description.

Convenience methods for adding callback-events, oop-callback-events, and autonomous events are also included; these are `add-callback-event`, `add-object-callback-event`, and `add-autonomous-event`. All of these take a time as their first argument (after the DESL instance, that is). For all of these, as well as `add-event`, there are the following keyword arguments:

- `:description` – the event description
- `:creator` – who created this event; can be left nil
- `:from-now?` – if set, the event's time is take to be relative to the current time; this is the default
- `:from-start?` – if set, then the time is considered to be relative to the start of the simulation; if neither is set, then it is taken to be an absolute time.⁸
- `:time-units` – specifies the units of the time

Additionally, `add-callback-event` requires a function (not a method) to be specified as its `:callback` argument. This will be called when the event occurs, with the arguments event time and event description. `add-object-callback-event` requires a callback *method* be given as `:callback`'s value, as well as the object that method is to be applied to, as `:object`'s value. This method will have three arguments, the object, the event time, and the description. `add-autonomous-event` takes a `:handler` keyword argument that should have the value of a method of the `autonomous-event` class or a subclass of it. It also takes a `:class` argument that allows you to specify your own event class, which should be a subclass of `autonomous-event`.

```
[00132] (defmethod add-event ((self desl) event &key &allow-other-keys)
[00133]   (with-slots (event-queue events-in current-time) self
[00134]     (with-slots (time description) event
[00135]       (cond
[00136]         ((< time current-time)
[00137]          (fmsg "Can't enqueue an event prior to the current time (~s, ~s, ~s)."
[00138]               event time description)
[00139]          nil)
[00140]         (t
[00141]          (enqueue event-queue event)
[00142]          (incf events-in
[00143]              event))))))
[00144]
[00145] (defmethod add-event ((self desl) (time number)
[00146]                       &key (description (newSymbol 'event))
[00147]                             (creator 'automatic)
[00148]                             (from-now? t)
[00149]                             (from-start? nil)
[00150]                             (callback nil)
[00151]                             (callback-object nil)
[00152]                             (handler nil)
[00153]                             (class 'autonomous-event)
[00154]                             time-units)
[00155]   (cond
[00156]     (handler
[00157]      (add-autonomous-event self time
[00158]                            :handler handler
[00159]                            :class class
[00160]                            :creator creator
[00161]                            :simulator self
[00162]                            :description description
[00163]                            :from-start? from-start?
[00164]                            :from-now? from-now?
[00165]                            :time-units time-units
[00166]                            ))
[00167]     ((null callback)
[00168]      (add-event self (make-instance 'DESL-event
[00169]                                     :creator creator
[00170]                                     :simulator self
[00171]                                     :time (convert-incoming-time self
```

⁸Note: This should be changed in a later version to use the `time-as-absolute` setting.

```

[00172]                                     time :unit time-units
[00173]                                     :from-now? from-now?
[00174]                                     :from-start? from-start?)
[00175]                                     :description description)))
[00176] (callback-object
[00177]   (add-object-callback-event self
[00178]     time
[00179]     :callback callback
[00180]     :object callback-object
[00181]     :creator creator
[00182]     :simulator self
[00183]     :description description
[00184]     :from-start? from-start?
[00185]     :from-now? from-now?
[00186]     :time-units time-units
[00187]   ))
[00188] (t
[00189]   (add-callback-event self time
[00190]     :callback callback
[00191]     :creator creator
[00192]     :simulator self
[00193]     :description description
[00194]     :from-start? from-start?
[00195]     :from-now? from-now?
[00196]     :time-units time-units
[00197]   )))
[00198]
[00199] (defmethod add-callback-event ((self DESL) time &key
[00200]   (description (newSymbol 'callback-event))
[00201]   (creator 'automatic)
[00202]   (from-now? t)
[00203]   (from-start? nil)
[00204]   (callback nil)
[00205]   time-units
[00206]   simulator)
[00207] (add-event self
[00208]   (make-instance 'callback-event
[00209]     :time (convert-incoming-time self time :unit time-units
[00210]       :from-now? from-now?
[00211]       :from-start? from-start?)
[00212]     :callback callback
[00213]     :creator creator
[00214]     :simulator self
[00215]     :description description)))
[00216]
[00217] (defmethod add-object-callback-event ((self DESL) time &key
[00218]   (description (newSymbol 'callback-event))
[00219]   (callback nil)
[00220]   (object nil)
[00221]   (creator 'automatic)
[00222]   (from-now? t)
[00223]   (from-start? nil)
[00224]   time-units
[00225]   simulator)
[00226] (add-event self (make-instance 'OOP-callback-event
[00227]   :time (convert-incoming-time self time :unit time-units
[00228]     :from-now? from-now?

```

```

[00229]                                     :from-start? from-start?)
[00230]             :description description
[00231]             :callback callback
[00232]             :object object
[00233]             :creator creator
[00234]             :simulator self)))
[00235]
[00236] (defmethod add-autonomous-event ((self DESL) time &key
[00237]                                   description
[00238]                                   (class 'autonomous-event)
[00239]                                   (handler 'handler)
[00240]                                   (creator 'automatic)
[00241]                                   (from-now? t)
[00242]                                   (from-start? nil)
[00243]                                   time-units
[00244]                                   simulator)
[00245]   (add-event self
[00246]     (make-instance class
[00247]       :time (convert-incoming-time self time :unit time-units
[00248]         :from-now? from-now?
[00249]         :from-start? from-start?)
[00250]       :description description
[00251]       :handler handler
[00252]       :creator creator
[00253]       :simulator self)))
[00254]

```

Method peek(desl)

Method dequeue(desl)

The `peek` method returns the next event in DESL's queue without dequeuing it. The `dequeue` method takes the event off the queue.

```

[00255] (defmethod peek ((self DESL))
[00256]   (with-slots (event-queue) self
[00257]     (peek event-queue)))
[00258]
[00259] (defmethod dequeue ((self DESL))
[00260]   (with-slots (event-queue events-out current-time ignore-deleted-events) self

```

This needs to be changed when priority queues, etc., handle deleted events correctly.

```

[00261]     (when (typep event-queue 'calendar-queue)
[00262]       (setf (slot-value event-queue 'ignore-deleted-events)
[00263]         ignore-deleted-events))
[00264]     (unless (empty? event-queue)
[00265]       (let ((event (dequeue event-queue)))
[00266]         (with-slots (time description) event
[00267]           (dfmsg "[~s: dequeing ~s (time=~s)]" self event time)

```

If there was an event, then we need to do a few things: update the event out count, for one, and also update the current time to move ahead to this event.

```

[00268]         (incf events-out)
[00269]         (setq current-time time))
[00270]       event))))
[00271]

```

Method `process-next-event(DES�)`

Method `process-next-events(DES�) :number :all`

The `process-next-event` method dequeues an event and does whatever needs to be done with it, depending on the kind of event it is. `process-next-events` will process more than one. If the `number` keyword is set, then that's how many is processed; if `all` is set, then all events are processed. `number` takes precedence over `[[all]]`, if both are set. These methods rely on `process-event` to actually handle the event.

Method `process-event(DES�,event)`

This method does whatever is necessary, based on the kind of event it is passed. Events come in several flavors, including those that you can define yourself (based on the pre-defined ones, most likely). At the most basic, an event that is created by the system is of type `DES�-event`. When one of these is dequeued, just the description is returned as the primary value, with the time and the event itself as additional values (returned via `values`).

There are two kinds of events that, when processed, call functions or methods the user specifies. The first of these, `callback-event`, has a function as its callback; this function is called with the time and description from the event itself. This kind of event is created by `DES�` for the user. The second, `oop-callback-event`, is for use by OO programs whose programmer would like the callback to be a method of some object. In this case, the `callback` method is applied to `object`, with the time and description as additional arguments. The third kind of event, `autonomous-event`, is one that handles itself. When this is encountered, `DES�` calls the event handler method of that event, which is named in the `handler` instance variable. This kind of event is for users to base their own event classes on. In these cases, what is returned from `process-event` is simply whatever these functions/methods return.

```
[00272] (defmethod process-next-event ((self DESL))
[00273]   (let ((event (dequeue self)))
[00274]     (dfmsg "[~s: dequeued event ~s at ~s; processing]"
[00275]           self (event-description event)
[00276]           (simtime-to-string self :time (event-time event)))
[00277]     (vfmsg "~a - Event ~s occurred."
[00278]           (current-timestring self :as-units nil)
[00279]           (event-description event))
[00280]
[00281]     (process-event self event)))
[00282]
[00283] (defmethod process-event ((self DESL) (event event))
[00284]   (cond
[00285]     ((null event)
[00286]      (dfmsg "[~s: trying to process null event]" self)
[00287]      nil)
[00288]     ((slot-value event 'deleted)
[00289]      (vfmsg "Event ~s was deleted; not processing." event)
[00290]      nil)
[00291]     ((typep event 'oop-callback-event)
[00292]      (with-slots (object callback time description) event
[00293]        (dfmsg "[~s: processing OOP-callback-event ~s (~s, ~s) at ~s]" self event
[00294]              object description time)
[00295]        (funcall callback object time description)))
[00296]     ((typep event 'callback-event)
[00297]      (with-slots (callback time description) event
[00298]        (dfmsg "[~s: processing callback-event ~s (~s) at ~s]"
[00299]              self event description time)
[00300]        (funcall callback time description)))
[00301]     ((typep event 'autonomous-event)
[00302]      (with-slots (handler time description) event
[00303]        (dfmsg "[~s: processing autonomous-event ~s (~s) at ~s]"
[00304]              self event description time)
```

```

[00305]         (funcall handler event)))
[00306]     ((typep event 'DESL-event)
[00307]      (values (event-description event) (event-time event) event))
[00308]     (t
[00309]      (with-slots (time description) event
[00310]        (warn "Unknown kind of event ~s encountered (time=~s, desc=~s)."
[00311]              event time description)
[00312]        (values nil (event-time event) event))))))
[00313]
[00314] (defmethod process-next-events ((self DESL) &key number all)
[00315]   (with-slots (event-queue) self
[00316]     (cond
[00317]       ((numberp number)
[00318]        (setq all nil))
[00319]       (number (setq number 0))                ;invalid -- skip entirely
[00320]       (all
[00321]        (setq number 1)))                    ;we'll not decrement it.
[00322]
[00323]       (loop until (or (empty? event-queue)
[00324]                       (<= number 0))
[00325]         do
[00326]           (process-next-event self)
[00327]           (unless all
[00328]             (decf number))))))
[00329]

```

Method `run-one(DESL)`

Method `run-n(DESL,n)`

Method `run-all(DESL)`

Method `run-until(DESL,time,:time-units,:from-now?,:from-start?)`

These methods are the primary user interface to DESL for getting events processed. `run-one` will run a single event, `run-n` will run n events, `run-all` will run until there are no more events, and `run-until` will process events until `time` occurs, inclusive. For `run-until`, `time` is considered to be in units of `time-units` or, if unspecified, the value stored in the `time-units` instance variable. `from-start?` and `from-now?` have the same meaning as in the `convert-incoming-time` method.

NOTE: in a future version of DESL, `run-continuously` should be defined, to associate a Lisp process with the DESL instance that will run whenever there is an event in the queue.

```

[00330] (defmethod run-one ((self DESL))
[00331]   (process-next-event self))
[00332]
[00333] (defmethod run-n ((self DESL) n)
[00334]   (process-next-events self :number n))
[00335]
[00336] (defmethod run-all ((self DESL))
[00337]   (process-next-events self :all t))
[00338]
[00339] (defmethod run-until ((self desl) time &key from-now? from-start? time-units)
[00340]   (with-slots (current-time time-is-absolute start-time) self
[00341]     (setq time
[00342]       (convert-incoming-time self time :unit time-units
[00343]                               :from-now? from-now?
[00344]                               :from-start? from-start?))
[00345]
[00346]     (loop until (or (empty? self)

```

```

[00347]             (> (event-time (peek self))
[00348]                 time))
[00349]         do
[00350]           (process-next-event self)))
[00351]

```

Occasionally, we need to remove an event. We may want to, for example:

- remove the first n events (`remove-first-event`)
- remove all events prior to some time (`remove-events-before`), possibly the current time (`remove-past-events`); and
- remove a particular event, either specified as the event itself or as a time and (possibly) description (`remove-event`).

```

[00352] (defmethod remove-first-event ((self DESL) &optional (n 1))
[00353]   (loop for i from 1 to n do
[00354]     (dequeue self)))
[00355]
[00356] (defmethod remove-past-events ((self DESL))
[00357]   (with-slots (current-time) self
[00358]     (let ((ctime current-time))
[00359]       (remove-events-before self ctime nil)

```

Make sure that `current-time`, which is updated by `dequeue`, is still set correctly:

```

[00360]     (setq current-time ctime))))
[00361]
[00362] (defmethod remove-events-before ((self DESL) time &optional (inclusive? t))
[00363]   (with-slots (event-queue) self
[00364]     (let ((pred (if inclusive? #'<=
[00365]                       #'<)))
[00366]       (loop
[00367]         while (and (not (empty? event-queue))
[00368]                   (funcall pred (event-time (peek self)) time))
[00369]           do (dequeue self))))))
[00370]
[00371] (defmethod remove-event ((self DESL) (event event) &optional ignore)
[00372]   (declare (ignore ignore))
[00373]   (with-slots (event-queue) self
[00374]     (remove-event event-queue event)))
[00375]
[00376] (defmethod remove-event ((self DESL) (time number) &optional description)
[00377]   (with-slots (event-queue) self
[00378]     (remove-event event-queue time description)))
[00379]

```

Method `empty?(DESL)`

This returns `t` if there are no events in the queue.

```

[00380] (defmethod empty? ((self DESL))
[00381]   (with-slots (event-queue) self
[00382]     (empty? event-queue)))
[00383]

```

Method `dump-events(desl,:sort,:limit)`

The `dump-events` method returns all the events in the queue. If `sort` is set (the default), then the events are sorted by increasing time. If `limit` is set, then that number of events is returned.

```
[00384] (defmethod dump-events ((self DESL) &key (sort t) limit)
[00385]   (with-slots (event-queue) self
[00386]     (dump-events event-queue :sort sort :limit limit)))
[00387]
```

Method find-event-by-description(desl,description,:time,:test,:all)

Method find-event-by-time(desl,time,:description,:test,:all)

These methods are used to find events matching some criteria. The first looks for one or more (depending on the setting of `all`) events matching `description` when compared by `test`, which defaults to `equal`. If `time` is specified, then all matching events must happen according to that constraint. As discussed elsewhere, `time` can either be a number, meaning the time must be equal to that, or a constraint. The constraint looks just like a corresponding Lisp Boolean or relational expression, with the event's time left out. So

```
(or (< 15) (and (> 20) (<= 30)))
```

means the event should occur prior to time 15 or between 20 and 30 (inclusive of 30), that is, corresponding to the Lisp expression

```
(or (< event-time 15) (and (> event-time 20) (<= event-time 30)))
```

`find-event-by-time` looks first for events meeting the `time` criterion, then winnows those by description.

```
[00388] (defmethod find-event-by-description ((self desl)
[00389]                                          description &key time (test #'equal)
[00390]                                          all)
[00391]   (with-slots (event-queue) self
[00392]     (find-event-by-description event-queue description :time time
[00393]                               :test test :all all)))
[00394]
[00395] (defmethod find-event-by-time ((self desl) time
[00396]                                 &key description (test #'equal)
[00397]                                 all)
[00398]   (with-slots (event-queue) self
[00399]     (find-event-by-time event-queue time :description description
[00400]                         :test test :all all)))
[00401]
[00402]
```

Method delete-event(DES�,:event,:description,:time,:all,:test,:really-remove)

This method deletes an event (or more than one, if `all` is specified) from DESL's `event-queue`. An event can be specified as an event object, passed as the value of `event`. Alternatively, the user can specify the time, the description, or both. (If `event` is a non-event object and `description` isn't set, then `event` is taken to be the event description.) `test` specifies which function to use to compare `description` to the event's description. `time` is really a time description, as described in the documentation for `find-event-by-time` and `find-event-by-description`; see those functions for details.

Unless `really-remove` is set, then the events will just be marked (using their `deleted` instance variables) as deleted, but left in the queue. If `really-remove` is set, then the events are removed from the queue, which will take longer but will save (usually a little) space.

```
[00403] (defmethod delete-event ((self DESL) &key event time description
[00404]                               all
[00405]                               (test #'equal)
[00406]                               really-remove)
[00407]   (with-slots (event-queue) self
[00408]     (delete-event event-queue :event event :time time :description description
```

```

[00409]             :test test
[00410]             :all all :really-remove really-remove)))
[00411]
[00412]

```

8.4 Time-related methods and functions

Method `set-start-time`

This sets the start time for the simulation. If `now` is set, the current system (universal) time is used. If `zero` is set, then the time is set to 0. Otherwise, a time is composed based on the settings of the other keyword arguments. The result will be a universal time (offset from 1/1/1900). If any of the keyword arguments are missing, then the method tries to do something intelligent. So, for example, if `month` is set, but not `year`, then the current year is used.

```

[00413] (defmethod set-start-time ((self DESL) &key now second minute hour day
[00414]                               month year raw zero)
[00415]   (with-slots (start-time current-time) self
[00416]     (mbind (s m h d mo y)
[00417]       (decode-universal-time (get-universal-time))
[00418]       (declare (ignore s))
[00419]       (cond
[00420]         (now
[00421]          (setq start-time (get-universal-time)))
[00422]         (raw
[00423]          (setq start-time raw))
[00424]         (zero
[00425]          (setq start-time 0))
[00426]         ((and second (not (or year month day hour minute)))
[00427]          (setq start-time (encode-universal-time
[00428]                             second
[00429]                             m h d mo y)))
[00430]         ((and minute (not (or hour day month year)))
[00431]          (setq start-time (encode-universal-time
[00432]                             (or second 0) minute h d mo y)))
[00433]         ((and hour (not (or day month year)))
[00434]          (setq start-time (encode-universal-time
[00435]                             (or second 0)
[00436]                             (or minute 0)
[00437]                             hour d mo y)))
[00438]         ((and day (not (or month year)))
[00439]          (setq start-time (encode-universal-time
[00440]                             (or second 0)
[00441]                             (or minute 0)
[00442]                             (or hour 0)
[00443]                             day mo y)))
[00444]         ((and month (not year))
[00445]          (setq start-time (encode-universal-time
[00446]                             (or second 0)
[00447]                             (or minute 0)
[00448]                             (or hour 0)
[00449]                             (or day 1)
[00450]                             month y)))
[00451]         (year
[00452]          (setq start-time (encode-universal-time
[00453]                             (or second 0)
[00454]                             (or minute 0)

```

```

[00455]             (or minute 0)
[00456]             (or day 1)
[00457]             (or month 1)
[00458]             year)))
[00459]     ((or year month day hour minute second)
[00460]      (setq start-time (encode-universal-time
[00461]                       (or second 0)
[00462]                       (or minute 0)
[00463]                       (or minute 0)
[00464]                       (or day 1)
[00465]                       (or month 1)
[00466]                       (or year y))))
[00467]     (t
[00468]      (setq start-time (get-universal-time))))
[00469]     (setq current-time start-time)
[00470]     (initialize-queue self))
[00471]

```

Function `time-delta`

Events will refer to actual universal times, but often we will need to specify some event happening some time in the future as a time relative to the current time. So we'll need a method that can figure out that time, as well as more generic methods to compute the interval from some particular time.

`time-delta` will determine a universal time interval based on its parameters of seconds, minutes, hours, etc. Note that `months`, although allowed, defaults to 30 days, and a year is considered to be 365 days (no leap years considered).

```

[00472] (defun time-delta (&key seconds minutes hours
[00473]                   days weeks months years milliseconds)
[00474]   (let ((interval 0.0d0))
[00475]     (when seconds
[00476]       (incf interval seconds))
[00477]     (when minutes
[00478]       (incf interval (m->s minutes)))
[00479]     (when hours
[00480]       (incf interval (h->s hours)))
[00481]     (when days
[00482]       (incf interval (d->s days)))
[00483]     (when weeks
[00484]       (incf interval (w->s weeks)))
[00485]     (when months
[00486]       (incf interval (mo->s months)))
[00487]     (when years
[00488]       (incf interval (y->s years)))
[00489]     (when (and milliseconds (not (zerop milliseconds)))
[00490]       (setq interval (+ interval (/ milliseconds 1000.0))))
[00491]     interval))
[00492]
[00493]
[00494] (defun m->s (m)
[00495]   (* m 60))
[00496]
[00497] (defun h->s (h)
[00498]   (* h 3600))
[00499]
[00500] (defun d->s (d)
[00501]   (* d 24 3600))
[00502]

```

```

[00503] (defun w->s (w)
[00504]   (* w 7 (d->s 1)))
[00505]
[00506] (defun mo->s (mo &optional (month-length 30))
[00507]   (* mo month-length (d->s 1)))
[00508]
[00509] (defun y->s (y)                                ;ignoring leap years!
[00510]   (* y 365 (d->s 1)))
[00511]

```

Method now-plus-interval

This method will give a new universal time based on the current time, but in the future as specified by the keyword arguments `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`.

```

[00512] (defmethod now-plus-interval ((self DESL)
[00513]   &key seconds minutes hours days weeks months years)
[00514]   (with-slots (current-time) self
[00515]     (+ current-time (time-delta :seconds seconds
[00516]                               :minutes minutes
[00517]                               :hours hours
[00518]                               :days days
[00519]                               :weeks weeks
[00520]                               :months months
[00521]                               :years years))))
[00522]

```

Method current-time(desl)

This method returns the current time as a Unix (universal) time.

```

[00523] (defmethod current-time ((self desl))
[00524]   (with-slots (current-time) self
[00525]     current-time))
[00526]

```

Method current-timestring(DES�,:as-units)

The `current-timestring` method will return the current time, as a string. `as-units`, if true, will cause the value to be reported as a number of some units, otherwise, a prettier string will be returned (see `simtime-to-string`). Which units is determined by the value of `as-units`: if it is a unit (as specified in the `valid-units` instance variable), then those units are used; if it is just `t`, then the value of the instance variable `time-units` is used. If `absolute` is set, then the time is taken to be a universal time (well, with milliseconds); if not, then if `relative` is set, the time is shown relative to `start-time` (i.e., `current-time` is subtracted from `start-time`). If neither is set, then the value of the `time-is-absolute` instance variable is used.

```

[00527] (defmethod current-timestring ((self desl) &key (as-units t)
[00528]   absolute
[00529]   relative)
[00530]   (with-slots (current-time) self
[00531]     (simtime-to-string self
[00532]       :time current-time
[00533]       :as-units as-units
[00534]       :absolute absolute
[00535]       :relative relative)))
[00536]

```

Method `simtime(desl,:milliseconds,:seconds,...,:from-now?,:from-start?)`

This method returns a simulation time, as specified by the various time keyword parameters: `milliseconds`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`. If `from-now?` is set, then the resulting time is added to the `current-time` instance variable; if `from-start?` is set, it is added to the `start-time` variable. These are mutually exclusive.

```
[00537] (defmethod simtime ((self DESL) &key milliseconds seconds minutes
[00538]                               hours days weeks months years
[00539]                               from-start?
[00540]                               from-now?)
[00541]   (with-slots (current-time start-time) self
[00542]     (let ((time-as-seconds (time-delta
[00543]                               :milliseconds milliseconds
[00544]                               :seconds seconds
[00545]                               :minutes minutes
[00546]                               :days days
[00547]                               :weeks weeks
[00548]                               :years years
[00549]                               :months months)))
[00550]       (cond
[00551]         (from-now?
[00552]          (+ time-as-seconds current-time))
[00553]         (from-start?
[00554]          (+ time-as-seconds start-time))
[00555]         (t time-as-seconds))))))
[00556]
```

Method `simtime-to-string(DES�,:time,:relative-to-start)`

The `simtime-to-string` method pretty-prints a time. If `time` is not set, then `current-time` is used. If `relative-to-start` is set (the default), then the time is computed by subtracting `time` (or `current-time`) from `start-time`, else the time is considered a universal time. Time is printed as something like 1y 2w 3d 04:05:06.003, and is shortened if weeks, years, and/or days aren't present.

```
[00557] (defmethod simtime-to-string ((self desl) &key time
[00558]                               as-units
[00559]                               absolute
[00560]                               relative)
[00561]   (with-slots (current-time time-is-absolute time-units start-time) self
```

If not time is set, use `current-time`:

```
[00562]   (unless time
[00563]     (setq time current-time))
```

Now determine whether to use absolute or relative (to the start) times based on `absolute` and `relative`, along with the `time-is-absolute` instance variable.

```
[00564]   (when (or relative (and (not absolute)
[00565]                             (not time-is-absolute)))
[00566]     (setq time (- time start-time)))
[00567]
[00568]   (cond
[00569]     (as-units
[00570]      (format nil "~3,3f" (time-to-units self time :units as-units)))
[00571]     (t
[00572]      (mbind (ms s m h d w y)
[00573]              (parse-time self time)
[00574]              (format nil "~a~a~a~2,'Od:~2,'Od:~2,'Od~a"
[00575]                      (if (zerop y)
```

```

[00576]         ""
[00577]         (format nil "~sy " y))
[00578]         (if (zerop w)
[00579]             ""
[00580]             (format nil "~sw " w))
[00581]         (if (zerop d)
[00582]             ""
[00583]             (format nil "~sd " d))
[00584]         h m s
[00585]         (if (< ms 0.0005)
[00586]             ""
[00587]             (format nil "~3,3f" ms)))))))))
[00588]

```

Method `time-to-units(DES�,time,:units)`

The basic time unit for DESL is the second; times are generally then compatible with universal times, except that DESL stores the time number as a double float, so that milliseconds can be represented, as well. The `time-to-units` method will convert `time` to a number of `units`; if `units` is nil or `t`, then it will use the value of the instance variable `time-units`.

```

[00589] (defmethod time-to-units ((self DESL) time &key units)
[00590]   (with-slots (time-units valid-time-units) self
[00591]     (unless (member units valid-time-units)
[00592]       (setq units time-units))
[00593]     (float
[00594]      (/ time
[00595]         (case units
[00596]           (:years (y->s 1))
[00597]           (:weeks (w->s 1))
[00598]           (:days (d->s 1))
[00599]           (:hours (h->s 1))
[00600]           (:minutes (m->s 1))
[00601]           (:seconds 1)
[00602]           (:milliseconds 0.001d0))))))
[00603]
[00604] (defmethod parse-time ((self DESL) time)
[00605]   (let (milli s m h d w y left)
[00606]     (msetq (time milli)
[00607]       (floor time))
[00608]     (msetq (y left)
[00609]       (floor time (y->s 1)))
[00610]     (msetq (w left)
[00611]       (floor left (w->s 1)))
[00612]     (msetq (d left)
[00613]       (floor left (d->s 1)))
[00614]     (msetq (h left)
[00615]       (floor left (h->s 1)))
[00616]     (msetq (m s)
[00617]       (floor left (m->s 1)))
[00618]     (values milli s m h d w y))
[00619]

```

Method `convert-incoming-time(DES�,time,:unit,:from-now?,:from-start?)`

This method will convert a time to the universal time needed by DESL. If `unit` is specified, then `time` is assumed to be in those units. `from-now?` and `from-start?` specify if `time` is to be considered relative to the current time or the start time. If `unit` is omitted, then the `time-units` instance variable is used; if the

from-xxx? are omitted, then the value of the instance variable `time-is-absolute` is used to determine if time is considered an absolute (i.e., universal) time or a time relative to `start-time`.

```
[00620] (defmethod convert-incoming-time ((self DESL) time &key unit from-now? from-start?)
[00621]   (with-slots (current-time start-time time-units valid-time-units) self
[00622]     (unless unit
[00623]       (setq unit time-units))
[00624]     (cond
[00625]       ((null time) (error "Can't have nil time."))
[00626]       ((not (member unit valid-time-units))
[00627]        (error "Invalid time unit ~s specified." unit))
[00628]       (t
[00629]        (setq time (time-delta :milliseconds (if (eql unit :milliseconds)
[00630]                                                  time 0)
[00631]                          :seconds (if (eql unit :seconds)
[00632]                                        time 0)
[00633]                          :minutes (if (eql unit :minutes)
[00634]                                        time 0)
[00635]                          :hours (if (eql unit :hours)
[00636]                                       time 0)
[00637]                          :days (if (eql unit :days)
[00638]                                       time 0)
[00639]                          :weeks (if (eql unit :weeks)
[00640]                                       time 0)
[00641]                          :years (if (eql unit :years)
[00642]                                       time 0))))
[00643]       (cond
[00644]         (from-now? (+ time current-time))
[00645]         (from-start? (+ time start-time))
[00646]         (t time))))))
[00647]
```

8.5 Miscellaneous Functions

Method `foutput(calendar-queue,&rest l)`

Method `set-verbosity(calendar-queue,&rest l)`

Method `verbosity(calendar-queue,&rest l)`

DESL makes use of the MaineSAIL utilities, in particular its “message handler” facilities. This is a way to potentially give each object in an OOP its own output manager, called a message handler (an instance of `message-handler`). Messages are output via macros such as `fmsg`, `vfmsg`, `dfmsg`, and `vdmsg`, which output normal verbosity, verbose, debugging, and verbose debugging messages, respectively. Verbosity of a message handler is controlled via the `set-verbosity` method. See the documentation for the utilities for more information.

`foutput`, `set-verbosity`, and `verbosity` are defined for DESL, as well, as pass-through methods that must exist if `fmsg`, etc., are to use the `*message-handler*` instance variable of `calendar-queue` rather than the global one. This allows messages for the calendar queue to be controlled (e.g., their verbosity) separately from other things that use `fmsg` and their kin.

```
[00648] (defmethod foutput ((self DESL) &rest l)
[00649]   (with-slots (*message-handler*) self
[00650]     (apply #'foutput (cons *message-handler* l))))
[00651]
[00652] (defmethod set-verbosity ((self DESL) &optional (verb-level *msg*))
[00653]   (with-slots (*message-handler*) self
[00654]     (set-verbosity *message-handler* verb-level)))
```

```

[00655]
[00656] (defmethod verbosity ((self DESL))
[00657]   (with-slots (*message-handler*) self
[00658]     (verbosity *message-handler*)))
[00659]
[00660] (defmethod print-events ((self desl) &key (print-each t)
[00661]                           (limit 100)
[00662]                           (sort t) short)
[00663]   (with-slots (event-queue) self
[00664]     (print-events event-queue :print-each print-each
[00665]                       :limit limit
[00666]                       :sort sort
[00667]                       :short short)))
[00668]

```

References

- Brown, R. (1988). Calendar queues: A fast, $(o)(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM (CACM)*, 31(10):1220–1227.
- Grimm, V. & Railsback, S. F. (2005). *Individual-based Modeling and Ecology*. Princeton Series in Theoretical and Computational Biology. Princeton University Press, Princeton, NJ.
- Turner, R. M. (2010). Literate programming in Lisp (LP/Lisp). Technical Report 2007–02, Department of Computer Science, University of Maine, 5752 Neville Hall, Orono, ME 04469–5752.

Index

- add-autonomous-event (method; code line 231), 10
- add-callback-event (method; code line 194), 9
- add-event (method; code line 125), 8
- add-event (method; code line 138), 8
- add-object-callback-event (method; code line 212), 9
- autonomous-event (class; code line 114), 7

- callback-event (class; code line 94), 6
- convert-incoming-time (method; code line 666), 21
- current-time (method; code line 549), 18
- current-timestring (method; code line 553), 18

- d->s (function; code line 525), 17
- delete-event (method; code line 424), 14
- dequeue (method; code line 256), 10
- DESL (class; code line 1), 4
- DESL-event (class; code line 88), 6
- dump-events (method; code line 400), 14

- empty? (method; code line 393), 13

- find-event-by-description (method; code line 407), 14
- find-event-by-time (method; code line 414), 14
- foutput (method; code line 696), 21

- h->s (function; code line 522), 17
- handle (method; code line 119), 7

- initialize-instance (method; code line 48), 5
- initialize-queue (method; code line 52), 5

- m->s (function; code line 519), 17
- make-event-queue (method; code line 72), 5
- mo->s (function; code line 531), 17

- now-plus-interval (method; code line 538), 18

- OOP-callback-event (class; code line 101), 6

- parse-time (method; code line 646), 20
- peek (method; code line 252), 10
- print-events (method; code line 710), 21
- process-event (method; code line 287), 11
- process-next-event (method; code line 276), 11
- process-next-events (method; code line 319), 12

- remove-event (method; code line 382), 13
- remove-event (method; code line 387), 13
- remove-events-before (method; code line 372), 13
- remove-first-event (method; code line 360), 13
- remove-past-events (method; code line 364), 13
- run-all (method; code line 342), 12
- run-n (method; code line 339), 12
- run-one (method; code line 336), 12
- run-until (method; code line 345), 13

- set-start-time (method; code line 436), 16
- set-verbosity (method; code line 700), 21
- simtime (method; code line 565), 18
- simtime-to-string (method; code line 587), 19

- time-delta (function; code line 497), 17
- time-to-units (method; code line 623), 20

- verbosity (method; code line 704), 21

- w->s (function; code line 528), 17

- y->s (function; code line 534), 17