# The Markov Classes Package

## Instructions for Users

Version 1.3

Hans Ekkehard Plesser (October 2002)

# Chapter 1

# Generalities

## 1.1 Prerequisites

The source code an documentation for the MarkovClasses package is available from `http://arken.nlh.no/~itfhep/software`. Please download `MarkovClasses.tgz` from there and unpack it. The package assumes that you have the GCC compiler (`http://gcc.gnu.org`).

## 1.2 Introduction

The theory behind the logarithmic classes is explained in Fricke and Wendt (1995, Sec. IV), the theory behind the discrete classes in Plesser and Wendt (1996). Preprints of both papers are included in the MarkovClasses package. The instructions given here assume that you are familiar with the theory behind the logarithmic or discrete class algorithms.

These instructions should provide you as a programmer of simulation software with the extra information necessary to successfully employ the LogclassDisclass routines in your code. At first, an introduction to the data structure is given and afterwards the functions are explained. At the end of these instructions you will find a sample program that demonstrates the use of the Package.

First of all, we need to clarify some terms that will be used throughout this document and the source code, but that is not always consistent with Fricke & Wendt.

- **event**: An *event* is a single step of a Markov process. When simulating conformation changes of polymers as a Markov process, e.g., an event may be the twisting of one binding by a certain angle. In order to simulate such a Markov process, each *possible event* must be represented, i.e. it must be assigned a probability and information about what kind of event it is, like the number of a binding and the tilting angle. This information is stored

in so–called *event descriptors*. Obviously, there must be a descriptor for each possible event. In many cases, though, it might be easier to call an entire set of possible Markov steps an event, as it is done by Fricke&Wendt. They call all steps that originate from a *cell* of their grid one *event*, so *cell* and *event* become synonymous; thus, Logclass/Disclass just takes care of selecting cells in which things are to happen and the user–program decides afterwards which of the possible steps (e.g. diffusion to any next neighbor, chemical reaction with another molecule in the same cell) takes place. In this case the reactivity of the event/cell is the sum of the rates of all possible steps originating from the cell.

- **ued**: **User's event descriptor**, represents each possible event within the user–program's data. These descriptors contain the information required to "execute" an event, e.g. a list of neighboring cells. It must contain a pointer to its event's **led**.
- **led**: Logclass **event descriptor**, represents a possible event within the Logclass data and contains the information required to select an event for "execution". It contains a pointer to its event's **ued**.
- **ded**: Disclass **event descriptor**, equivalent to **led**.
- **rate**: the probability for an event to happen per unit time, assuming an exponential FPT–distribution with mean $1/r$. Class and total rate are the sums of their elements' rates. You may occasionally find the term "reactivity" instead of rate.
- **nullclass**: Besides the usual classes of possible events, there is a pseudo-class `nullclass`, to which all impossible events are assigned, e.g. immobilized adatoms in a study of surface dynamics. To allow for efficient simulation, you may assign any numerical value (which is not an index of a usual class) as index of the `nullclass`.

## 1.3 Some words of caution

**Do not rely on anything** stated in this document or in the comments included in the source code. We have tried our best, but we do not guarantee for anything! The code included has been tested extensively on various problems, but there may still be bugs around.

**Think twice** before you change any of the code included, because even though it might be flawed, chances are pretty good to screw up some of the delicate memory management included! For the same reason, do not manipulate the data stored in the Logclass/Disclass data structures in any way but calling the appropriate Logclass/Disclass functions.

**Do not use** variable names beginning with `dlc_` or `dc_` in your user–program as all of Logclass'/Disclass' type and function names begin with these magic letters (as an abbreviation for dual logarithmic/discrete classes).

# Chapter 2

# Logarithmic Classes

## 2.1 Data structures

As explained by Fricke&Wendt, a program using LOGCLASS requires two separate sets of data, one provided and maintained by the user–program (**ued**s) and one supplied by the LOGCLASS routines, consisting of **led**s and some overhead. The connection between those sets of data is made by pointers connecting the two descriptors belonging to each possible event.

The LOGCLASS data structure has three "layers". On top is a single variable of type `dlc_global` that contains — besides some other information — pointers to the array of class descriptors. The latter form the second layer, an array containing one element of type `dlc_class` per logarithmic class. Each of these class descriptors points to the class' event–array $F_z[]$, containing one **led** (of type `dlc_event`) per event in the class plus "free" **led**s to be assigned to future members of the class.

### 2.1.1 User-program's data

```
typedef struct { /* whatever you want */
                 dlc_event *led;
               } user_event_descriptor;
```

As the name says, it is up to you to design this data type with just a few things to keep in mind. You *must* include a pointer of type `dlc_event*` in the structure, so the **ued** can point to its associated **led**; return values of some LOGCLASS functions have to be assigned to this pointer (see below), but it must not be manipulated otherwise. The rate of the event described should not be stored in the **ued**, as it is already present in the **led** and may be read using `ued->led->r`. For easy access and flexibility, an array of **ued**s should be allocated using `calloc`; after the **ued**s have been assigned **led**s using `dlc_enter`, the **ued**s *must not* be moved in memory, because this would destroy the link **ued** $\leftrightarrow$ **led**.

### 2.1.2 dlc_global

```
typedef struct DLC_GLOBAL { dlc_class *cbeg, *cend, *first;
                            int       min_class, num_classes;
                            size_t    max_events;
                            double    r, eps;
                            long int  number_of_reorgs;
                            void      (*event_moved)(dlc_event *);
                            char      err_file[NAME_MAX];
                            void      (*err_proc)();
                            void       *err_ptr;
                          } dlc_global;
```

This data structure contains the global data needed to access all of the class and event data. It has to be initialized by `dlc_init` (see below) before the simulation starts. The user–program should allocate memory for this structure and provide a pointer to it, since all LOGCLASS functions expect a pointer to this structure as first argument. An appropriate definition would thus be

```
dlc_global                    global_data,
          *global_data_ptr = &global_data;
```

This data structure is essentially private, i.e. *the user should keep her/his hands off its contents*, except for members `r`, `eps` and `number_of_reorgs`, which may be *read* by the user–program. `r` is the total rate which determines the random time step according to

$$\Delta t = -\frac{1}{r}\log(1 - \mathbf{rnd})$$

where **rnd** is a uniformly distributed random number, $\mathbf{rnd} \in [0, 1)$, or, coded in C using Fricke/Knuth's random generator,

```
simulation_time += exp_rand55() / global_data_ptr->r;  .
```

`eps` is the lowest rate not treated as zero and might be used to count the number of thus "impossible" events. `number_of_reorgs` counts the reorganizations of LOGCLASS' event data structure and thus helps to monitor performance.

### 2.1.3 dlc_class

```
typedef struct DLC_CLASS{ dlc_event        *bot, *top;
                          double           r, r_max, eps;
                          struct DLC_CLASS *next;
                        } dlc_class;
```

This data structure is private, i.e. *the user should keep her/his hands off its contents.*

### 2.1.4 dlc_event

```
typedef struct DLC_EVENT{ void   *ued;
                          double r;
                        } dlc_event;
```

For each Markov event with a non–zero probability (`dlc_event.r >= dlc_global.eps`) to occur, there is one **led** of type `dlc_event`. The member `ued` provides the link **led** → **ued**, which is maintained by the Logclass routines. `ued` is of type `void*`, so that the actual **ued** structure in the user–program can have any name and contents. When using `dlc_event.ued`, it has to be typecast properly. The user–program *must not* manipulate the **led** data.

## 2.2 Public functions

All functions contained in the logclass package take as their first argument a pointer to the logclass global data structure `dlc_glob *dlc` that contains all relevant information after initialization by `dlc_init`. This argument will not be discussed again in the descriptions of the individual functions.

### 2.2.1 dlc_init

```
void dlc_init(dlc_global *dlc, size_t num_leds, double min_reac,
              double max_reac, long rand_seed, void (*event_moved)(),
              char err_file[], void (*err_proc)(), void *err_ptr);
```

This function *must* be called before any other Logclass function, as it allocates memory and sets up links. The parameters passed to `dlc_init` are checked to some degree and an error message is issued prior to program termination if inappropriate parameters are given or not enough memory is available. Parameters are

| | | |
|---|---|---|
| `dlc_glob` | `*dlc` | pointer to Logclass global data, which has to be defined by the user program (see above, `dlc_glob`) |
| `size_t` | `num_leds` | number of **led**s to be allocated |
| `double` | `min_reac` | expected lowest rate |
| `double` | `max_reac` | expected highest rate |
| `long` | `rand_seed` | random seed to be used |
| `void` | `(*event_moved)()` | pointer to function updating link **ued** → **led** |
| `char` | `err_file[]` | file to which error messages shall be written |
| `void` | `(*err_proc)()` | pointer to function which is to be called after an error message has been issued |
| `void` | `*err_ptr` | err_proc will be called with argument err_ptr, argument, so the user can pass a reference to his/her data to the "wrap up routine"; if you have no such pointer, pass NULL. |

- `num_leds` should be the number of possible events, e.g. the number of cells in a grid or the number of bindings in a polymer, times a "memory factor"

(usu. $2-5$), depending on the memory available. The extra memory is spread over all classes (uniformly at the start, proportional to current size by later re–organizations), so that events that change classes due to modified rate can simply be placed at the end of the array of **led**s of their new class. If too few **led**s are allocated, "busy" classes will frequently be filled to their limit, leading to time–consuming memory reorganizations that move all **led**s. Thus, you should increase your "memory factor" if your program requires more than one reorganization per $10^5$ Markov events (you can get the number of reorganizations from `dlc_global`).

- `min_reac` and `max_reac` give lower and upper limits to the range of rates to be handled; additional classes are added at both ends of the range according to the preprocessor constants `DLC_EXTRA_BOTTOM/TOP`, so the range that can be handled is given by $\epsilon \le r < r_{max}$ with

$$\begin{aligned} \epsilon &:= 2^{\lfloor \log_2 \texttt{min\_reac} \rfloor - \texttt{DLC\_EXTRA\_BOTTOM} - 1} \\ r_{max} &:= 2^{\lfloor \log_2 \texttt{max\_reac} \rfloor + \texttt{DLC\_EXTRA\_TOP} + 1} \end{aligned}$$

  where $\lfloor x \rfloor$ is the largest integer not greater than $x$.

  To avoid round–off errors, the ratio between minimum and maximum rate should be greater than machine precision, i.e. $\epsilon / r_{max} \ge \texttt{DBL\_EPSILON} \approx 10^{-16}$ (for 64–bit doubles).

  $r_{max}$ is an absolute upper limit: as soon as a single event reaches a rate greater than $r_{max}$, the simulation is terminated, because no class is available to store this event. Thus, if anything goes wrong at the upper end, you will notice. The lower limit is much more dangerous in this respect, as all events with rates below $\epsilon$ (stored in `global_data.eps`) will be considered impossible, i.e. their rate will be considered as 0 and so they do not get a **led** and therefore they cannot be selected to happen (for details, see `dlc_safechange`). This might be sensible, as no particles can diffuse out of or react within an empty cell, e.g., so the cell's rate actually is zero. If there is a large number of events with $0 < r < \epsilon$, though, the simulation might produce wrong results, as the total rate becomes too small and thus time steps too large and events with high rates are selected too often. If in doubt, you might want to keep track of the number of "impossible" events using `global_data.eps`.

- `rand_seed` is used as seed for the random generator. If it is set to `0L`, the clock will be used as seed.

- `event_moved()` is explained below. When calling `dlc_init`, include the function name in the parameter list *without* parentheses.

- `err_file[]` must not be an empty string, but a valid file name.

- `err_proc()` is a function that is called after the LOGCLASS' own error handler has written its message to `err_file`. In this way, control is returned to the user program prior to program termination, so that data

may be saved. The function passed as `err_proc` must be of type `void` and may not take any arguments. It may contain its own `exit` call, but that is not required. Important: `err_proc` is *not* called if an error occurs in `dlc_init`, since no simulation data is around then.

`dlc_init` may terminate issuing one of the following error messages:

- `Invalid parameter to dlc_init: min_reac > 0 req ...`
  The minimum rate must be *greater* than zero
- `Invalid parameters to dlc_init: min_reac = ...`
  $\epsilon/r_{max} \geq$ `DBL_EPSILON` must be fulfilled
- `Initialization warning: On the average you have less than two events per class.`
  The considerable administrative overhead required for the logarithmic classes makes sense only for large systems.
- `Initialization error: unable to allocate memory ...`
  Too bad, ey .... Looks like you'll have to get along with a smaller system or dig up some bucks to by more RAM.

### 2.2.2 event_moved

The links **ued** → **led** and **led** → **ued** between user–progam and LOGCLASS data must be maintained under all circumstances. This is easy for **led** → **ued**, as the **ued**s are not moved in memory. It is also quite simple to maintain **ued** → **led** when an event is entered or moved using `dlc_enter` and `dlc_safechange`, as the pointer to the new **led** is returned. Problems occur once the array of **led**s has to be re–organized and when a **led** has to be moved to fill a gap in a **led**–array $F_z[]$. In principle, the link **ued** → **led** could be maintained easily even in this case writing

```
moved_events_new_led->ued->led = moved_events_new_led;
```

where `moved_events_new_led` is of type `dlc_event*`. Unfortunately, this requires the **ued** to contain an element `led`, interfering with the idea that LOGCLASS should be as autonomous and flexible as possible.

To circumvent this problem, the user–program has to provide its own function to update the **ued** → **led** link. This function must

- be of type `void`,
- take as a single argument of type `dlc_event*` the pointer to the **led** that changed,
- "follow" the **led** → **ued** link and update **ued** → **led**.

Thus, it will usually look like

```
void my_u2l_updating_function(dlc_event *new_led)
{
  ( (user_event_descriptor *) new_led->ued )->led = new_led;
}
```

### 2.2.3 dlc_clear

```
void dlc_clear(dlc_global *dlc)
```

This function returns all dynamic memory allocated by `dlc_init` to the system. It should be called before leaving the simulation program. Its only argument is a pointer to the Logclass global data structure.

### 2.2.4 dlc_enter

```
dlc_event *dlc_enter(dlc_global *dlc, void *ued, double r)
```

This function should be called after Logclass has been initialized using `dlc_init` and the system's **ued**s have been prepared. At this stage, the events are placed in the logarithmic classes calling `dlc_enter` once for each **ued**: the function determines the event's class according to its rate `r`, assigns a **led** to the event and creates the link **led** → **ued**. It returns the pointer to the **led**, so the link **ued** → **led** can be set at once using

```
ued->led = dlc_enter(global_data, ued, rate( ..., ued));
```

where `rate( ..., ued)` is a function provided by the user program and returning the rate of the event described by **ued** as a `double` value.

### 2.2.5 dlc_rand

```
dlc_event *dlc_rand(dlc_global *dlc)
```

This function picks one event by random (called `re`), according to the rates of the individual events, see Fricke&Wendt. It returns the pointer to the **led** of the event (`*re`).

### 2.2.6 dlc_safechange

```
dlc_event *dlc_safechange(dlc_global *dlc, dlc_event *led, void *ued,
                          double r)
```

This function is used to move the **led**s of those events whose rates `r` have changed as side–effect of the execution of the event selected by `dlc_rand` (call once for each event affected).

Arguments are

| | | |
|---|---|---|
| `dlc_global` | `*dlc` | see above |
| `dlc_event` | `*led` | pointer to the **led** of the event |
| `void` | `*ued` | pointer to the **ued** of the event |
| `double` | `r` | *new* reactivity of the event |

The function returns the pointer to the event's new **led**, required to update the link **ued** → **led**, compare `dlc_enter`; **led** → **ued** is maintained by the function.

If the event's new rate `r` is less than $\epsilon$ (`global_data.eps`, see `dlc_init`), its rate is set to zero and it is not assigned a **led**: `dlc_safechange` returns a `NULL` pointer. As the event does no longer have a **led**, it cannot be selected by `dlc_rand`, i.e. it cannot "happen". It might be, though, that its rate is increased beyond $\epsilon$ later on as a side–effect of another event. In this case, `dlc_safechange` puts the event back to the logarithmic classes, provides a new **led**, and re–creates the link **led** → **ued**.

## 2.3 Sample Program

See `dlc_demo.c` in the Markov Classes package.

# Chapter 3

# Discrete Classes

## 3.1 Data structures

As explained by Fricke & Wendt, a program using DISCLASS requires two separate sets of data, one provided and maintained by the user program (**ued**s) and one supplied by the DISCLASS routines, consisting of **ded**s and some overhead. The connection between those sets of data is made by pointers connecting the two descriptors belonging to each possible event.

The DISCLASS data structure has three "layers". On top is a single variable of type `dc_global` that contains — besides some other information — pointers to the array of class descriptors. The latter form the second layer, an array containing one element of type `dc_class` per discrete class. Each of these class descriptors points to the class' event–array $F_z[]$, containing one **ded** (of type `dc_event`) per event in the class plus "free" **ded**s to be assigned to future members of the class.

### 3.1.1 User-program's data

```
typedef struct {
                /* whatever you want */
                dc_event *ded;
              } user_event_descriptor;
```

As the name says, it is up to you to design this data type with just a few things to keep in mind. You *must* include a pointer of type `dc_event*` in the structure, so the **ued** can point to its associated **ded**; return values of some DISCLASS functions have to be assigned to this pointer (see below), but it must not be manipulated otherwise. The class to which the event belongs should not be stored in the **ued**, as it is already present in the **ded** and may be read using `ued->ded->ci`. For easy access and flexibility, an array of **ued**s should be allocated using `calloc`; after the **ued**s have been assigned **ded**s using `dc_store`,

the **ued**s **must not be moved in memory**, because this would destroy the links **ued** $\leftrightarrow$ **ded**.

### 3.1.2 dc_global

```
typedef struct DC_GLOBAL { dc_class  *cbeg, *cend, *first;
                           int       num_classes, nullclass;
                           size_t    max_events;
                           double    r;
                           long int  number_of_reorgs;
                           void      (*event_moved)(dc_event *);
                           char      err_file[NAME_MAX];
                           void      (*err_proc)();
                           void      *err_ptr;
                         } dc_global;
```

This data structure contains the global data needed to access all of the class and event data. It has to be initialized by `dc_init` (see below) before the simulation starts. The user program should allocate memory for this structure and provide a pointer to it, since all DISCLASS functions expect a pointer to this structure as their first argument. An appropriate definition would be

```
dc_global                    global_data,
         *global_data_ptr = &global_data;
```

This data structure is essentially private, i.e. **the user should keep her/his hands off its contents**, except for members `r` and `number_of_reorgs`, which may be **read** by the user program.

`r` is the total rate which determines the random time step according to

$$\Delta t = -\frac{1}{r}\log(1 - \mathbf{rnd})$$

where **rnd** is a uniformly distributed random number, **rnd** $\in [0, 1)$. Using Fricke/Knuth's random generator, the appropriate `C` code is

```
simulation_time += exp_rand55() / global_data_ptr->r;  .
```

`number_of_reorgs` counts the reorganizations of DISCLASS' event data structure and thus helps to monitor performance.

### 3.1.3 dc_class

```
typedef struct DC_CLASS{ dc_event         *bot, *top;
                         double           r, r_ind;
                         struct DC_CLASS  *next;
                       } dc_class;
```

This data structure is private, i.e. **the user should keep her/his hands off its contents**.

### 3.1.4 dc_event

```
typedef struct DC_EVENT{ void   *ued;
                         int    ci;
                       } dc_event;
```

For each Markov event with a non–zero probability to occur (class index `ci` is not `nullclass`), there is one **ded** of type `dc_event`. The member `ued` provides the link **led** → **ued**, which is maintained by the Disclass routines. `ued` is of type `void*`, so that the actual **ued** structure in the user program can have any name and contents. When using `dc_event.ued`, it has to be typecast properly. The user program **must not manipulate** the **ded** data.

## 3.2 Public functions

All Disclass functions take as their first argument a pointer to the Disclass global data structure `dc_global *dc` that contains all relevant information after initialization by `dc_init`. This argument will not be discussed again in the descriptions of the individual functions.

### 3.2.1 dc_init

```
void dc_init(dc_global *dc, size_t num_deds, int num_classes,
             double *class_r_ind, int nullclass, long rand_seed,
             void (*event_moved)(), char err_file[], void (*err_proc),
             void *err_ptr);
```

This function **must be called before** any other Disclass function, as it allocates memory and sets up links. The parameters passed to `dc_init` are checked to some degree and an error message is issued prior to program termination if inappropriate parameters are given or not enough memory is available. Parameters are

| | | |
|---|---|---|
| `dc_global` | `*dc` | pointer to Disclass global data,which has to be defined by the user program (see above, `dc_glob`) |
| `size_t` | `num_deds` | number of **ded**s to be allocated |
| `int` | `num_classes` | number of discrete classes |
| `double` | `*class_r_ind` | pointer to array of class' individual event rates |
| `int` | `nullclass` | pseudo-class for impossible events |
| `long` | `rand_seed` | random seed to be used |
| `void` | `(*event_moved)()` | pointer to function updating link **ued** → **led** |
| `char` | `err_file[]` | file to which error messages shall be written |
| `void` | `(*err_proc)()` | pointer to function which is to be called after an error message has been issued |
| `void` | `*err_ptr` | pointer to user program's data to be passed to err_proc |

`num_deds` should be the number of possible events, e.g. the number of cells in a grid or the number of bindings in a polymer, times a "memory factor" (usu. $2 - 5$), depending on the memory available. The extra memory is spread over all classes (uniformly at the start, proportional to current size by later re-organizations), so that events that change classes due to modified rate can simply be placed at the end of the array of **ded**s of their new class. If too few **ded**s are allocated, "busy" classes will frequently be filled to their limit, leading to time-consuming memory reorganizations that move all **ded**s. Thus, you should increase your "memory factor" if your program requires more than one reorganization per $10^5$ Markov events (you can get the number of reorganizations from `dc_global`).

`num_classes` is the number of classes describing your system. The pseudo-class for impossible events is **not** included in the number.

`class_r_ind` is the pointer to the first element of an array with **exactly** `num_classes` entries. Each entry gives the rate of an **individual** event in the corresponding class. The contents of the array is copied by `dc_init`, so the array may be removed after `dc_init` was executed.

`nullclass` the class index of the pseudo-class of impossible events. You can choose this value as it fits your simulation needs. E.g., you might classify the adatoms in a simulation by the number of next neighbors, yielding $0 \ldots 3$ as "real" class indices and `nullclass` $= 4$ as impossible pseudo-class (compare Plesser & Wendt).

`rand_seed` is used as seed for the random generator. When using the Fricke/Knuth random generator, setting `rand_seed` to `0L` will be use the clock as seed.

`event_moved()` is explained below. When calling `dc_init`, include the function name in the parameter list **without** parentheses.

`err_file[]` must not be an empty string, but a valid file name. All error messages will be sent to this file.

`err_proc()` is a function that is called after the DISCLASS' own error handler has written its message to `err_file`. In this way, control is returned to the user program prior to program termination, so that data may be saved. The function passed as `err_proc` must be of type `void` and may take zero or one argument. It may contain its own `exit` call, but that is not required.

`err_ptr` is a pointer to an arbitrary data structure and will be passed on to `err_proc` upon error. If you pass `NULL`, `err_proc` will be called without argument.

`dc_init` may terminate issuing the following error message:
`Initialization error: unable to allocate memory ...`
Try reducing the memory requirement by asking for a smaller `num_ded`s, but you should not reduce your "memory factor" below 2. In the worst case, stick with smaller systems or buy more RAM.

### 3.2.2 event_moved

The links **ued** → **led** and **led** → **ued** between user progam and DISCLASS data must be maintained under all circumstances. This is easy for **led** → **ued**, as the **ued**s are not moved in memory. It is also quite simple to maintain **ued** → **led** when an event is entered or moved using `dc_store`, as the pointer to the new **ded** is returned. Problems occur once the array of **ded**s has to be re-organized and when a **ded** has to be moved to fill a gap in a **ded**-array $F_z[]$. In principle, the link **ued** → **led** could be maintained easily even in this case writing

```
moved_events_new_ded->ued->ded = moved_events_new_ded;
```

where `moved_events_new_ded` is of type `dc_event*`. Unfortunately, this requires the **ued** to contain an element `ded`, interfering with the idea that DISCLASS should be as autonomous and flexible as possible.

To circumvent this problem, the user program has to provide its own "linking" function to update the **ued** → **led** link. This function must

- be of type `void`,
- take as a single argument of type `dc_event*` the pointer to the **ded** that changed,
- "follow" the **led** → **ued** link and update **ued** → **led**.

Thus, it will usually look like

```
void my_ued2ded_updating_function(dc_event *new_ded)
{
  ( (user_event_descriptor *) new_ded->ued )->ded = new_ded;
}
```

### 3.2.3 dc_rand

```
dc_event *dc_rand(dc_global *dc)
```

This function picks one event by random (called `re`), according to the rates of the individual events, see Plesser & Wendt. It returns the pointer to the **ded** of the event (`*re`).

### 3.2.4 dc_store

```
dc_event *dc_store(dc_global *dc, void *ued, int new_ci)
```

This function is used enter events into the DISCLASS data structures at the beginning of the simulation. Arguments are

| | | |
|---|---|---|
| `dc_global` | `*dc` | see above |
| `void` | `*ued` | pointer to event's **ued** |
| `int` | `new_ci` | index of event's class |

The event is deleted from its current class and inserted into the new one; the link **led** → **ued** is created by the function. The opposite link **ued** → **led** needs to be set up by the user program using the return value of `dc_store`, e.g.

```
ued->ded = dc_store(global_data, ued, ci)
```

Note that impossible events should not be entered into the data structures. `dc_store` should be called by the user program **upon initialization only**. Any subsequent updating should be done through `dc_move`.

### 3.2.5 dc_move

```
dc_event *dc_move(dc_global *dc, dc_event *ded, void *ued, int new_ci)
```

This function is used to move the **ded**s of those events whose rates (may) have changed upon execution of an event. Arguments are

| | | |
|---|---|---|
| `dc_global` | `*dc` | see above |
| `dc_event` | `*ded` | pointer to event's **ded** |
| `void` | `*ued` | pointer to event's **ued** |
| `int` | `new_ci` | index of event's **new** class; pass `nullclass` if event has become impossible |

The event is deleted from its current class and inserted into the new one; the link **led** → **ued** is maintained by the function. The opposite link **ued** → **led** needs to be maintained by the user program using the return value of `dc_move`, e.g.

```
ued->ded = dc_move(global_data, ued->ded, ued, new_ci)
```

If `new_ci == nullclass`, the event is just deleted from its old class and assigned to the pseudo-class, i.e. it is not assigned a new **ded**. Thus, the function returns `NULL` in this case.

### 3.2.6 dc_clear

```
void dc_clear(dc_global *dc)
```

This function returns all dynamic memory allocated by `dc_init` to the system. It should be called before leaving the simulation program. Its only argument is a pointer to the DISCLASS global data structure.

## 3.3 Using a different random generator

To use a different random generator than the one by Fricke/Knuth, take a look at the following lines in `dc.1.1.h` (approx. line 70)

```
/* modify the following four lines to use your own random generator          */
#include "random.h"   /* random generator by Fricke/Knuth                    */
#define seedrand(seed) (init_rand55(seed))   /* seeding command              */
#define longrand(max)  (lrand55(max))        /* long rnd, 0 <= ... < max     */
#define doublerand()   (drand55())           /* double rnd, 0<= ... < 1      */
```

The `#include` command should be replaced with the appropriate include for your own random generator. The macros **must** be defined in such a way that

`seedrand(seed)` seeds the random generator, where `seed` is a `long`. For consistency, `seedrand(0L)` should seed using the clock.

`longrand(max)` returns a uniformly distributed random number of type `long` in the interval $[0, \texttt{max})$.

`doublerand()` returns a uniformly distributed random number of type `double` in the interval $[0, 1)$.

# Bibliography

Fricke, T. and D. Wendt (1995). The markoff automaton—a new algorithm for simulating the time evolution of large stochastic dynamic systems. *Int J Mod Phys C 6*, 277–302.

Plesser, H. E. and D. Wendt (1996). A fast algorithm for high-dimensional markov processes with finite sets of transition rates. In *Proceedings of the 1996 International Symposium on Nonlinear Theory and its Applications (NOLTA '96)*, Kochi, Japan, pp. 249–252.