Uppaal DBM Library
Programmer's Reference

Alexandre David

October 23, 2006

Contents

Chapter 1

Introduction

Difference bound matrices (DBMs) are efficient data structures commonly used in verification timed automata [1]. UPPAAL is a verification tool for timed automata and uses this library for operating on DBMs. However, DBMs can only represent convex sets so the library provides access to federations as well, an arbitrary union of DBMs. The library architecture (Fig. 1.1) is as follows: A core of C functions for basic operations on DBMs serves as a basis for a C++ implementation of two main classes `dbm_t` and `fed_t` that implement DBMs and federations. The C++ API is developper friendly in the sense that memory allocation is hidden in the library and these structures can be manipulated as simple scalar types cheaply since the library implements reference counting and copy-on-write. A Ruby binding is available to access federations with different modules: `udbm` is the core module for federations, `udbm-gtk` is the module for the graphical viewer (based on Gtk), `udbm-sys` is a higher level abstraction on system of constraints where DBMs are entered only by means of constraints between clocks.

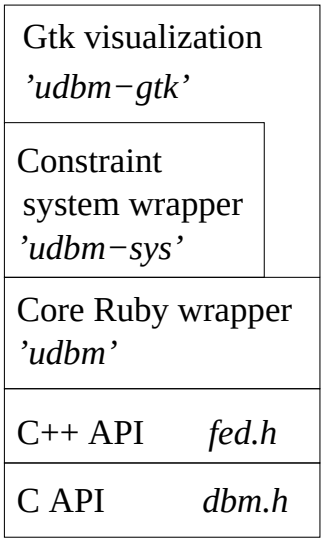

Figure 1.1: Architecture of the DBM library.

Chapter 2

C API

2.1 Header file `constraints.h`

2.1.1 Types

DBMs are matrices of clock constraints of the form $x_i - x_j < b_{ij}$ or $x_i - x_j \leq b_{ij}$. A clock constraint is internally encoded into the type `raw_t`. In contrast bounds are integers (`int32_t`).

The type `constraint_t` regroups the indices of clock constraints and the encoded constraint itself (bound + strict or non strict inequality). This type is also available with a constructor in C++.

```
typedef struct {
    index_t i,j;
    raw_t value;
} constraint_t;
```

Creation of constraints can be done with one of these two functions:

```
constraint_t dbm_constraint(index_t i, index_t j,
                            int32_t bound, strictness_t strictness);

constraint_t dbm_constraint2(index_t i, index_t j,
                             int32_t bound, BOOL isStrict);
```

The indices `i` and `j` correspond to the entry (i, j) in the DBM for the constraint. The bound is given followed by either a strictness type (see Constants) or a boolean that says if the constraint has a strict inequality or not.

2.1.2 Constants

`dbm_INFINITY`	The infinity bound.
`dbm_OVERFLOW`	Bound to test for overflow.
`dbm_LE_ZERO`	Constraint encoding of ≤ 0.
`dbm_LS_INFINITY`	Constraint encoding of $< \infty$.
`dbm_LS_OVERFLOW`	Constraint to test for overflow.
`dbm_STRICT`	Enum type for strict inequalities ($<$).
`dbm_WEAK`	Enum type for weak inequalities ($\leq$).

2.1.3 Conversion Functions

dbm_boundbool2raw

Synopsis: `raw_t dbm_boundbool2raw(int32_t bound, BOOL isStrict);`

Description: Convert a bound and a flag telling if the inequality is strict or not to an encoded constraint.

dbm_raw2bound

Synopsis: `int32_t dbm_raw2bound(raw_t raw);`

Description: Decode a constraint and return its bound.

dbm_strictRaw

Synopsis: `raw_t dbm_strictRaw(raw_t raw);`

Description: Make a constraint strict (strict inequality).

dbm_weakRaw

Synopsis: `raw_t dbm_weakRaw(raw_t raw);`

Description: Make a constraint weak (weak inequality).

dbm_raw2strict

Synopsis: `strictness_t dbm_raw2strict(raw_t raw);`

Description: Decode a constraint and return its strictness (enum type, see the constants in Subsection 2.1.2).

dbm_rawIsStrict

Synopsis: `BOOL dbm_rawIsStrict(raw_t raw);`

Description: Test if a constraint is strict – return TRUE ($<$) or FALSE ($\leq$).

dbm_rawIsWeak

Synopsis: `BOOL dbm_rawIsWeak(raw_t raw);`

Description: Test if a constraint is weak – return TRUE ($\leq$) or FALSE ($<$).

dbm_negStrict

Synopsis: `strictness_t dbm_negStrict(strictness_t strictness);`

Description: Negate the strictness of a constraint.

dbm_negRaw

Synopsis: `raw_t dbm_negRaw(raw_t c);`

Description: Negate a constraint, e.g., the negation of $< a$ is $\leq -a$ and the negation of $\leq a$ is $< -a$.

dbm_isValidRaw

Synopsis: `BOOL dbm_isValidRaw(raw_t x);`

Description: Test if a constraint is valie, i.e., it should not cause overflow in (addition) operations.

dbm_negConstraint

Synopsis: `constraint_t dbm_negConstraint(constraint_t c);`

Description: Negate a constraint and return its negation. The negation of $x_i - x_j < b_{ij}$ is $x_j - x_i \leq -b_{ij}$ and the negation of $x_i - x_j \leq b_{ij}$ is $x_j - x_i < -b_{ij}$.

dbm_areConstraintsEqual

Synopsis: `BOOL dbm_areConstraintsEqual(constraint_t c1, constraint_t c2);`

Description: Test if two constraints c1 and c2 are equal, which is same indices and same constraint value.

2.1.4 Addition of Constraints

dbm_addRawRaw

Synopsis: `raw_t dbm_addRawRaw(raw_t x, raw_t y);`

Description: Addition of constraints. This is useful for the shortest path computation. The constraints x or y may be infinity. Examples: $(\leq 2)+(\leq 1) = (\leq 3)$, $(\leq 2) + (< 1) = (\leq 3)$, $(< 2) + (< 1) = (< 3)$, $(<= 1) + (< \infty) = (< \infty)$.

dbm_addRawFinite

Synopsis: `raw_t dbm_addRawFinite(raw_t x, raw_t y);`

Description: Addition of constraints with y being finite (not dbm_LS_INFINITY).

dbm_addFiniteRaw

Synopsis: `raw_t dbm_addFiniteRaw(raw_t x, raw_t y);`

Description: Addition of constraints with x being finite (not dbm_LS_INFINITY).

dbm_addFiniteFinite

Synopsis: `raw_t dbm_addFiniteFinite(raw_t x, raw_t y);`

Description: Addition of constraints with x and y being finite (not dbm_LS_INFINITY).

dbm_addFiniteWeak

Synopsis: `raw_t dbm_addFiniteWeak(raw_t x, raw_t y);`

Description: Specialized addition of constraints with x and y being finite (not dbm_LS_INFINITY) and at least x or y being a weak constraint.

dbm_rawInc

Synopsis: `raw_t dbm_rawInc(raw_t c, raw_t i);`

Description: Increment a constraint by i with a safe test for infinity. Notice that the increment (i) is in raw_t format. Addition of constraints will result as $(\leq 0) + (\leq 0) = (\leq 0)$, whereas increment of constraints will result as $(\leq 0) + = 1 = (< 1)$ with 1 corresponding to internal encoding of (≤ 0).

dbm_rawDec

Synopsis: `raw_t dbm_rawDec(raw_t c, raw_t d);`

Description: Similarly to decrement a constraint. This has no effect if c is infinity.

2.2 Header file `dbm.h`

A dbm is defined as a squared matrix of `raw_t`. The type `raw_t` is the encoded clock constraint (see *constraints.h*). IMPORTANT: In the system, you will typically have clocks $x_1, x_2 ..., x_n$. The dbm has x_0 as the reference clock, hence the dimension is equal to $n+1$, which implies that we assume in all the functions that the dimension is **strictly** greater than 0.

The constraints of a DBM are refered as dbm[i,j] corresponding to the element dbm[i*dim+j] of the `raw_t` array. As a reminder, dbm[i,j] represents the constraint $x_i - x_j < b_{ij}$ or $x_i - x_j \leq b_{ij}$. The constraint encoding is described in *constraints.h*.

The C API does not support indirection table for clocks but the C++ API does. Dynamic mappings must be resolved before calling these functions. Be careful when dealing with operation that involve arrays of constraints (e.g., kExtrapolate). As a common assumption for all operations on DBM: $dim > 0$, which means at least the reference clock is in the DBM.

A non empty DBM means the represented zone is non empty, which is, for a closed dbm the diagonal elements are equal to 0. As a common assumption, all the DBMs taken as arguments are closed (the canonical form with the tightest constraints obtained by running the shortest path algorithm) and non empty. Resulting DBMs are either empty or closed and non empty.

The type `cindex_t` is used whenever an index or a dimension for a DBM is expected. An index is between 0 and $2^{16} - 1$ (even if the representation is on 32 bits). Obviously since DBMs are square matrices, this is a very reasonable limitation.

2.2.1 Basic Functions

dbm_init

Synopsis: `void dbm_init(raw_t *dbm, cindex_t dim);`

Description: Initialize a DBM `dbm` of dimension `dim` with ≤ 0 on the diagonal and the first row, and infinity everywhere else, which is an unconstrained DBM (with positive clocks).

dbm_zero

Synopsis: `void dbm_zero(raw_t *dbm, cindex_t dim);`

Description: Initialize a DBM `dbm` of dimension `dim` to the zero point (origin), which is set all the constraints to ≤ 0.

dbm_isEqualToInit

Synopsis: `BOOL dbm_isEqualToInit(const raw_t *dbm, cindex_t dim);`

Description: Test if the DBM `dbm` of dimension `dim` is equal to an unconstrained DBM (as written by dbm_init). Return TRUE if it is, FALSE otherwise.

dbm_isEqualToZero

Synopsis: `BOOL dbm_isEqualToZero(const raw_t *dbm, cindex_t dim);`

Description: Test if the DBM `dbm` of dimension `dim` is equal to the zero point (as written by dbm_zero). Return TRUE if it is, FALSE otherwise.

dbm_copy

Synopsis: `void dbm_copy(raw_t *dst, const raw_t *src, cindex_t dim);`

Description: Copy a DBM `src` of dimension `dim` to `dst`. Notice that the user has to make sure that `dst` points to an array `raw_t[dim*dim]`.

dbm_areEqual

Synopsis:
`BOOL dbm_areEqual(const raw_t *dbm1, const raw_t *dbm2, cindex_t dim);`

Description: Test if two DBMs are equal, i.e., are exactly identical, which makes sense for DBMs in closed form. It is also possible to call `dbm_relation` and test the result for `base_EQUAL` but this function is better and implements an optimistic test, which means that it will perform much better when DBMs are most often equal. This is particularly desirable when using hash tables to store DBMs.

dbm_hash

Synopsis: `uint32_t dbm_hash(const raw_t *dbm, cindex_t dim);`

Description: Compute a hash value for a DBM.

dbm_isPointIncluded

Synopsis:
```
BOOL dbm_isPointIncluded(const int32_t *pt, const raw_t *dbm,
cindex_t dim);
```

Description: Test if a discrete point (i.e., clock valuation) is included in the DBM (i.e., satisfies all the constraints of the DBM). Return TRUE if it is the case, FALSE otherwise. Note that the dimension of the point and the DBM must match. Also note that pt[0] should be 0 for this to be meaningful unless the user wants to include an offset.

dbm_isRealPointIncluded

Synopsis:
```
BOOL dbm_isRealPointIncluded(const double *pt, const raw_t *dbm,
cindex_t dim);
```

Description: Test if a real point (at the precision of double, of course) is included in the DBM. Return TRUE if it is the case, FALSE otherwise. Note that the dimension of the point and the DBM must match. Also note that pt[0] should be 0 for this to be meaningful unless the user wants to include an offset.

dbm_shrinkExpand

Synopsis:
```
cindex_t dbm_shrinkExpand(const raw_t *dbmSrc, raw_t *dbmDst,
cindex_t dimSrc, const uint32_t *bitSrc, const uint32_t *bitDst,
size_t bitSize, cindex_t *table);
```

Description: Shrink and expand a DBM, which is, resize it and copy, add, or remove clocks constraints corresponding to clocks that are copied, added, or removed. The arguments are:

- `dbmSrc`: The source DBM to resize. The necessary constraints will be copied and the DBM will not be modified.
- `dbmDst`: The destination DBM (enough space must be reserved) where to write the resulting DBM. It must be different from `dbmSrc`.
- `dimSrc`: The dimension of the source DBM. Note that the dimension of the destination is not given in argument because it is redundant with the information in `bitDst`. Instead, it is returned.
- `bitSrc` and `bitDst`: Bit tables to mark which clocks are represented in the source and the destination DBM. The idea is that the user has a number of global clocks but the DBMs are representing constraints on some of them only. The bit tables mark which clocks are used in the DBMs, i.e., if bit i is set then clock i is used. Of course clock i is represented by some index k in the DBM and a translation table (or mapping) is computed and returned as well. The first bit (bit 0) **must** be set (it is the reference clock). In addition, `bitSrc` and `bitDst` must be different (have different bits), which is, the function is supposed to be called if there is anything to

do since it is slightly more expensive than a simple copy (additional tests and computations are made).

- `bitSize`: The size of the bit table in integers (must be $\leq \lceil maxDim/32 \rceil$ where $maxDim$ is the maximal dimension, or total number of clocks.
- `table`: Where to write the resulting translation table (or mapping) for the result DBM.

The function returns the dimension of the resulting DBM (equal to the number of bits set in `bitDst`).

dbm_updateDBM

Synopsis:
`void dbm_updateDBM(raw_t *dbmDst, const raw_t *dbmSrc, cindex_t dimDst, cindex_t dimSrc, const cindex_t *cols);`

Description: Variant for resizing DBMs. Instead of giving arrays of bits, the user provides an array of the clocks wanted in the destination DBM (with the indices refering directly to the indices of the clocks of the source DBM). The resulting DBM is written in `dbmDst` (enough space must be allocated), the original is read from `dbmSrc`, the dimensions of the destination and the source, are `dimDst` and `dimSrc`. The table `cols` which clock to take for the destination and in which order: `cols[i]` tells which clock from the source to copy as clock i in the destination and if the special value `0` is used then a new unconstrained clock is added. Only the entries from 1 to dimDst-1 are meaningful with the entry at 0 being ignored since it corresponds to the reference clock that is always present (must always be 0).

dbm_swapClocks

Synopsis: `void dbm_swapClocks(raw_t *dbm, cindex_t dim, cindex_t x, cindex_t y);`

Description: Swap clocks `x` and `y`, which has the effect of swapping the corresponding constraints in the DBM `dbm` of dimension `dim`.

dbm_isDiagonalOK

Synopsis: `BOOL dbm_isDiagonalOK(const raw_t *dbm, cindex_t dim);`

Description: Test if the diagonal of a DBM is OK, which means that the constraints are either less than < 0 for an empty DBM or exactly ≤ 0 for a non empty DBM. This is useful only for debugging. Return TRUE if the diagonal is OK, FALSE otherwise.

dbm_isValid

Synopsis: `BOOL dbm_isValid(const raw_t *dbm, cindex_t dim);`

Description: Return TRUE if a DBM is closed, not empty, and the constraints in the first row are at most ≤ 0 (positive clocks), FALSE otherwise. It is not necessary to test for the diagonal separately. This is a very useful function to use in assertions, although it is expensive (cubic).

dbm_relation2string

Synopsis: `const char* dbm_relation2string(relation_t rel);`

Description: Convert a relation_t value to a meaningful string, which is useful for user feedback.

dbm_getMaxRange

Synopsis: `raw_t dbm_getMaxRange(const raw_t *dbm, cindex_t dim);`

Description: Compute the maximal range needed to store constraints of a DBM, excluding infinity (a special large value). This function is useful if the user intends to save DBMs on fewer than 32 bits.

2.2.2 DBM-DBM Operations

dbm_convexUnion

Synopsis:
```
void dbm_convexUnion(raw_t *dbm1, const raw_t *dbm2, cindex_t
dim);
```

Description: Compute the convex union of two DBMs. This implements "`*dbm1 += *dbm2`" where "+" refers to the convex union operator (used in the C++ API).

dbm_intersection

Synopsis:
```
BOOL dbm_intersection(raw_t *dbm1, const raw_t *dbm2, cindex_t
dim);
```

Description: Compute the intersection of two DBMs. This implements "`*dbm1 &= *dbm2`" where "&" refers to the intersection operator (used in the C++ API). The function returns TRUE if the resulting DBM is not empty, FALSE otherwise (and dbm1 is empty).

dbm_relaxedIntersection

Synopsis:
```
BOOL dbm_relaxedIntersection(raw_t *dbm1, const raw_t *dbm2,
cindex_t dim);
```

Description: Compute the intersection of two DBMs with their constraints relaxed. A relaxed constraint is a constraint made non strict if it is not infinity, e.g., $(< 3)^+ = (\leq 3)$. Infinity is always strict. The result is stored in dbm1 and operation corresponds to "`*dbm1 = (*dbm1)`$^+$ `& (*dbm2)`$^+$". The function returns TRUE if the resulting DBM is not empty, FALSE otherwise (and dbm1 is empty).

dbm_haveIntersection

Synopsis:
```
BOOL dbm_haveIntersection(const raw_t *dbm1, const raw_t *dbm2,
cindex_t dim);
```

Description: Test if two DBM have a non empty intersection. The test is approximate: The function returns FALSE if the intersection is empty for sure, or TRUE if the intersection is **maybe** not empty.

2.2.3 Constraining Operations

dbm_constrain

Synopsis:
```
BOOL dbm_constrain(raw_t *dbm, cindex_t dim, cindex_t i, cindex_t
j, raw_t constraint, uint32_t *untouched);
```

Description: This is the only function in the API that returns DBMs **that may not be closed**. Calls to `dbm_isEmpty` may return erroneous results unless dbm_closex is called before. This function is useful in the case where several constraints have to be applied on-the-fly to a DBM without knowing in advance all of them. The DBM `dbm` of dimension `dim` has its constraint $x_i - x_j$ constrained (or tightened if possible) with the constraint `constraint`. The function returns FALSE if the DBM is empty for sure or TRUE if it is **maybe** not empty. The array `untouched` is a bit array with `dim` bits, i.e., it is an array of $\lceil dim/32 \rceil$ `uint32_t`. It must be initialized to 0 for the first call and then it must not be modified between calls to dbm_constrain. This array is used to mark clocks that will be iterated over in the `dbm_closex` function to reduce, if possible, the iterations (each iteration is quadratic in function of dim).

dbm_constrainN

Synopsis:
```
BOOL dbm_constrainN(raw_t *dbm, cindex_t dim, const constraint_t
*constraints, size_t n);
```

Description: Constrain the DBM `dbm` of dimension `dim` with `n` constraints. The resulting DBM is either closed and not empty (and the function returns TRUE), or empty (and FALSE is returned). The constraint of the DBM are tightened if the argument constraints are tighter.

dbm_constrainIndexedN

Synopsis:
```
BOOL dbm_constrainIndexedN(raw_t *dbm, cindex_t dim, const
cindex_t *indexTable, const constraint_t *constraints, size_t n);
```

Description: Constrain the DBM `dbm` of dimension `dim` with `n` constraints, **but** constrain the constraints $x_{indexTable[i]} - x_{indexTable[j]}$ instead of $x_i - x_j$ as the previous function for all the constraints given in argument (a constraint has indices i and j, and a *value* field encoding the bound and the inequality). This function is useful in the case where DBMs are dynamically resized and a

global clock x may not correspond to the index x of the DBM but to the index $indexTable[x]$.

dbm_constrain1

Synopsis:
`BOOL dbm_constrain1(raw_t *dbm, cindex_t dim, cindex_t i, cindex_t j, raw_t constraint);`

Description: Constrain the DBM `dbm` of dimension `dim` with one constraint given in argument (for the constraint $x_i - x_j$). Return TRUE if the resulting DBM is not empty, FALSE otherwise.

dbm_constrainC

Synopsis:
`BOOL dbm_constrainC(raw_t *dbm, cindex_t dim, constraint_t c);`

Description: This is a wrapper for `dbm_constrain1(dbm, dim, c.i, c.j, c.value)`.

dbm_constrainClock

Synopsis:
`BOOL dbm_constrainClock(raw_t *dbm, cindex_t dim, cindex_t x, int32_t value);`

Description: Apply the constraint $x == value$ for a clock `x` to this DBM. This is the same as applying $x - x_0 \leq 0$ and $x_0 - x \leq 0$ to the DBM, except that this call is more efficient than making two consecutive calls. Return TRUE if the result is not empty, FALSE otherwise.

2.2.4 Standard Operations

dbm_up

Synopsis: `void dbm_up(raw_t *dbm, cindex_t dim);`

Description: Delay operation (future), also called strongest post-condition. The function sets the constraints $(i, 0)$ to infinity, i.e., $x_i - x_0 < \infty$.

dbm_down

Synopsis: `void dbm_down(raw_t *dbm, cindex_t dim);`

Description: Inverse delay operation (past), also called weakest pre-condition. The function removes the lower bounds of the clocks and update on-the-fly the closed form (some lower bounds may be induced by diagonal constraints). The clocks are still positive.

dbm_freeClock

Synopsis: `void dbm_freeClock(raw_t *dbm, cindex_t dim, cindex_t k);`

Description: Free a clock `k`, which is, remove all the constraints of this clock (except that the clock is still positive). This sets the constraints $x_i - x_k < \infty$ and $x_k - x_i < \infty \; \forall i$) except for $x_k - x_k \leq 0$ and $x_0 - x_k \leq 0$.

dbm_freeUp

Synopsis: `void dbm_freeUp(raw_t *dbm, cindex_t dim, cindex_t k);`

Description: Free the upper bounds of the clock `k`, i.e., set the constraints $x_k - x_i < \infty \; \forall i \neq k$).

dbm_freeAllUp

Synopsis: `void dbm_freeAllUp(raw_t *dbm, cindex_t dim);`

Description: Free the upper bounds for all the clocks (except the reference clock), i.e., set all the constraints to $x_i - x_j < \infty$ for $i > 0$ and $i \neq j$.

dbm_isFreedAllUp

Synopsis: `BOOL dbm_isFreedAllUp(const raw_t *dbm, cindex_t dim);`

Description: Test if calling `dbm_freeAllUp(dbm,dim)` has no effect on the DBM, i.e., if all the clocks have their upper bounds freed: Return TRUE if it is the case, FALSE otherwise.

dbm_freeDown

Synopsis: `void dbm_freeDown(raw_t *dbm, cindex_t dim, cindex_t k);`

Description: Free the lower bounds of the clock `k`, i.e., set the constraints (i, k) to ≤ 0 and tighten the DBM on-the-fly. In practice this means to set $dbm[i, k] := dbm[i, 0] \; \forall i \neq k$.

dbm_freeAllDown

Synopsis: `void dbm_freeAllDown(raw_t *dbm, cindex_t dim);`

Description: Free the lower bounds of all the clocks (except the reference clock), i.e., set $dbm[i, k] := dbm[i, 0] \; \forall i \neq k, \; \forall k > 0$.

dbm_testFreeAllDown

Synopsis: `BOOL dbm_testFreeAllDown(const raw_t *dbm, cindex_t dim);`

Description: Test if calling `dbm_freeAllDown(dbm,dim)` has no effect on the DBM, i.e., if all the clocks have their lower bounds freed: Return 0 if it is the case, or the value $(j << 16)|i$ where (i, j) corresponds to the constraint from where the DBM differs from the expected value.

dbm_satisfies

Synopsis:
`BOOL dbm_satisfies(const raw_t *dbm, cindex_t dim, cindex_t i, cindex_t j, raw_t constraint);`

Description: Test if the DBM `dbm` of dimension `dim` satisfies the constraint `constraint` ($x_i - x_j < b_{ij}$ or $x_i - x_j \leq b_{ij}$ encoded in `constraint`). This corresponds to applying the constraint to the DBM and checking if the result is not empty, except that we do not apply the constraint. The function must not be miss-used in testing consecutive constraints to conclude that the DBM satisfies their conjuction, which is wrong. However, it is right for a disjunction.

dbm_isEmpty

Synopsis: `BOOL dbm_isEmpty(const raw_t *dbm, cindex_t dim);`

Description: Test if a DBM is empty (return TRUE) or not (return FALSE). An empty DBM has a constraint stricly less than 0 (can be negative or can be just < 0), which results in no point satisfying the constraints. So either there is such a constraint on the diagonal (and the DBM is empty), or there is no such constraint but the DBM must be closed (canonical form) for the result to be right. Generally all the functions (except dbm_constrain) maintain this invariant.

dbm_close

Synopsis: `BOOL dbm_close(raw_t *dbm, cindex_t dim);`

Description: Apply the shortest path algorithm to the DBM to compute the tightest possible constraints. This is the canonical form of the DBMs and we refer to these DBM as being "closed". The result may be an empty DBM, hence the return value: Return TRUE if the DBM is not empty, FALSE otherwise. **Note**: This algorithm is cubic in function of dim!

dbm_isClosed

Synopsis: `BOOL dbm_isClosed(const raw_t *dbm, cindex_t dim);`

Description: Test if this DBM is in its "closed" form. This function is only useful for debugging or for assertions but be warned that it is as expensive as `dbm_close` (in fact more expensive because there is an internal allocation/copy/deallocation to test this). Return TRUE if the DBM is closed, FALSE otherwise.

dbm_closex

Synopsis:
`BOOL dbm_closex(raw_t *dbm, cindex_t dim, const uint32_t *touched);`

Description: This is a special version of `dbm_close` where only the clocks marked in `touched` will be tightened, i.e., if the bit k is set then the clock k is tightened. This is useful to reduce the cost of the close operation if we know that we need to tighten only certain clocks, which is the case when constraining DBMs. Return TRUE if the DBM is not empty, FALSE otherwise.

dbm_close1

Synopsis: `BOOL dbm_close1(raw_t *dbm, cindex_t dim, cindex_t k);`

Description: Special version of `dbm_closex` for tightening only one clock k. Return TRUE if the DBM is not empty, FALSE otherwise.

dbm_closeij

Synopsis:
`BOOL dbm_closeij(raw_t *dbm, cindex_t dim, cindex_t i, cindex_t j);`

Description: Special and more efficient version of `dbm_closex` for re-tightening a DBM after its constraint $x_i - x_j$ has been tightened! Note that this works only for $x_i - x_j$ and its cost is $dim + dim^2$ instead of $2 * dim^2$ if you call twice `dbm_close1` or if you call `dbm_closex` with the two bits i and j set.

dbm_tighten

Synopsis:
`void dbm_tighten(raw_t *dbm, cindex_t dim, cindex_t i, cindex_t j, raw_t c);`

Description: This is a shortcut for `dbm[i,j] := c` followed by a call to `dbm_closeij(dbm, dim, i, j)`. The function **assumes** that 1) it is a tightening ($c < dbm[i,j]$) and 2) the tightening results in a non empty DBM ($c + dbm[j,i] \geq 0$).

dbm_isUnbounded

Synopsis: `BOOL dbm_isUnbounded(const raw_t *dbm, cindex_t dim);`

Description: Test if a DBM is unbounded, i.e., if a point in the DBM can delay indefinitely. Return TRUE if it is, FALSE otherwise.

dbm_relation

Synopsis:
`relation_t dbm_relation(const raw_t *dbm1, const raw_t *dbm2, cindex_t dim);`

Description: Compute the relation between two DBMs `dbm1` and `dbm2` of dimension `dim` in the sense of set inclusion. The return value `relation_t` is an enumeration taking the values:

- `base_DIFFERENT` if dbm1 and dbm2 are not comparable,
- `base_SUPERSET` if dbm1 is a strict superset of dbm2,
- `base_SUBSET` if dbm1 is a strict subset of dbm2,
- `base_EQUAL` if dbm1 is equal to dbm2.

dbm_isSubsetEq

Synopsis:
`BOOL dbm_isSubsetEq(const raw_t *dbm1, const raw_t *dbm2, cindex_t dim);`

Description: Test if dbm1 is a subset (non strict) of dbm2. Return TRUE if it is, FALSE otherwise.

dbm_isSupersetEq

Synopsis:
`BOOL dbm_isSupersetEq(const raw_t *dbm1, const raw_t *dbm2, cindex_t dim);`

Description: Test if dbm1 is a superset (non strict) of dbm2. Return TRUE if it is, FALSE otherwise.

2.2.5 Update Operations

These operations correspond to updating a DBM to compute operations at the clock level, e.g., `x := 0` for a reset of the clock x to 0, `x := y` for copying the clock y to x, etc .... Using a specialized version is more efficient than the call to the general function `dbm_update`.

dbm_updateValue

Synopsis:
`void dbm_updateValue(raw_t *dim, cindex_t dim, cindex_t x, int32_t value);`

Description: Update a clock `x` to the value `value`, i.e., compute the operation `x := value` where value is a positive integer.

dbm_updateClock

Synopsis:
`void dbm_updateClock(raw_t *dbm, cindex_t dim, cindex_t x, cindex_t y);`

Description: Update a clock `x` to the clock `y`, i.e., compute the operation `x := y`.

dbm_updateIncrement

Synopsis:
`void dbm_updateIncrement(raw_t *dbm, cindex_t dim, cindex_t x, int32_t value);`

Description: Increment a clock `x` with `value`, i.e., compute the operation `x := x + value`. The value may be negative but the user has to make sure it is not too much negative, i.e., it will not result in a negative value for the clock `x`.

dbm_update

Synopsis:
`void dbm_update(raw_t *dbm, cindex_t dim, cindex_t x, cindex_t y, int32_t value);`

Description: This is a more general call to compute the operation `x := y + value`.

2.2.6 Relax Operations

Relaxing constraints means to make them less or equal (a.k.a. weak) when they are not $< \infty$. A constraint of the form $< b$ becomes $\leq b$. There are different relax operations to relax upper or lower bounds. The point is that they recompute the closed form on-the-fly whenever it is needed and the update is at most quadratic (and not cubic if running `dbm_close`). However, it is important to note that some functions may not be able to relax all the constraints they are suppose to relax if they are induced by other tighter constraints (a diagonal constraint may imply that another constraint must be strict).

dbm_relaxUpClock

Synopsis: `void dbm_relaxUpClock(raw_t *dbm, cindex_t dim, cindex_t x);`

Description: Relax the upper bounds of the clock `x`, i.e., make the constraints $x_k - x_i$ weak $\forall i$.

dbm_relaxDownClock

Synopsis: `void dbm_relaxDownClock(raw_t *dbm, cindex_t dim, cindex_t x);`

Description: Relax the lower bounds of the clock `x`, i.e., make the constraints $x_i - x_k$ weak $\forall i$.

dbm_relaxAll

Synopsis: `void dbm_relaxAll(raw_t *dbm, cindex_t dim);`

Description: Relax all the constraints (those that are not $< \infty$ of course).

dbm_relaxUp

Synopsis: `void dbm_relaxUp(raw_t *dbm, cindex_t dim);`

Description: Compute the smallest possible delay, which is the same as calling `dbm_relaxDown(dbm, dim, 0)` for the reference clock!

dbm_relaxDown

Synopsis: `void dbm_relaxDown(raw_t *dbm, cindex_t dim);`

Description: Compute the smallest possible inverse delay, which is the same as calling `dbm_relaxUp(dbm, dim, 0)` for the reference clock!

2.2.7 Extrapolation Operations

Extrapolations are approximation techniques to make sure that exploration algorithms will terminate. Depending on the models (in particular if diagonal constraints are used) the approximation can be exact, or not. The following different extrapolation algorithms are described in [2] and the names correspond to the algorithms of the paper. The arguments to the functions are the arrays of maximal constants (possibly different lower and upper maximal bounds). It is possible to give the special value `-dbm_INFINITY`, which has the effect to "free" the corresponding clock, which is a trick for implementing the so-called active clock reduction.

dbm_extrapolateMaxBounds

Synopsis:
```
void dbm_extrapolateMaxBounds(raw_t *dbm, cindex_t dim, const
int32_t *max);
```

Description: Classical extrapolation using maximal constants.

dbm_diagonalExtrapolateMaxBounds

Synopsis:
```
void dbm_diagonalExtrapolateMaxBounds(raw_t *dbm, cindex_t dim,
const int32_t *max);
```

Description: Diagonal extrapolation based on maximal bounds.

dbm_extrapolateLUBounds

Synopsis:
```
void dbm_extrapolateLUBounds(raw_t *dbm, cindex_t dim, const
int32_t *lower, const int32_t *upper);
```

Description: Extrapolation based on lower and upper bounds.

dbm_diagonalExtrapolateLUBounds

Synopsis:
```
void dbm_diagonalExtrapolateLUBounds(raw_t *dbm, cindex_t dim,
const int32_t *lower, const int32_t *upper);
```

Description: Diagonal extrapolation based on lower and upper bounds.

2.3 Header file `mingraph.h`

DBMs can be represented more compactly by removing redundant constraints and keeping only a minimal subset of constraints [4]. If we view DBMs as graphs we have constraints, the edges, defined between clocks, the vertices. This minimal representation is referred to as the *minimal graph*. The API gives access to the minimal graph directly and provides means of saving it somehow with different levels of compactness.

The idea is to have generic implementation that can be used with standard allocation schemes (malloc, new) and with custom allocators. This interface is in C to make it easy to wrap to other languages so we use a generic function to allocate memory. The type of this function is `int32_t* function(uint32_t size, void *data)`, where `size`e is the size in *int* to allocate, and it returns a pointer to a `int32_t[size]`, and `data` is other custom data. Possible wrappers are:

- for a custom allocator Alloc

```
int32_t *alloc(uint32_t size, void *data) {
    return ((Alloc*)data)->alloc(size);
 }
```

  defined as `base_allocate` in *base/DataAllocator.h*,

- for malloc

```
int32_t *alloc(uint32_t size, void *) {
    return (int32_t*) malloc(size*sizeof(int32_t));
}
```

  defined as `base_malloc` in *base/c_allocator.h*,

- for new

```
int32_t *alloc(uint32_t size, void *) {
    return new int32_t[size];
}
```

  defined as `base_new` in *base/DataAllocator.h*.

The allocator function and the custom data are packed together inside the `allocator_t` type.

dbm_writeToMinDBMWithOffset

Synopsis: `int32_t* dbm_writeToMinDBMWithOffset(const raw_t *dbm, cindex_t dim, BOOL minimizeGraph, BOOL tryConstraints16, alocator_t c_alloc, size_t offset);`

Description: Save a DBM in minimal representation. The API supports allocation of larger data structures than needed for the actual zone representation. When the `offset` argument is bigger than zero, `offset` extra integers are allocated and the zone is written with the given offset. Thus when *int32_t[data_size]* is needed to represent the reduced zone, an *int32_t* array of size `offset+data_size` is allocated. The first *offset* elements can be used by the caller. It is important to notice that the other functions typically expect a pointer to the actual zone data and not to the beginning of the allocated block. Thus in the following piece of code, most functions expect `mg and not` memory:

```
int32_t *memory = dbm_writeToMinDBMWithOffset(...);
mingraph_t mg = &memory[offset];
```

Notes: If `offset` is 0 and `dim` is 1, NULL may be returned. NULL is valid as an input to the other functions. It could be possible to send as argument the maximal value of the constraints that can be deduced from the maximal constants but this would tie the algorithm to the extrapolation.

The argument `minimizeGraph` activates the minimal graph reduction. The flag `tryConstraints16` enables saving the constraints on 16 bits (instead of 32). The allocation function conforms to the previous specification of memory allocation.

2.4 Header file `gen.h`

2.5 Header file `print.h`

Chapter 3

C++ API

3.1 Header file `constraints.h`

3.1.1 Type

When compiling with g++ the structure `constraint_t` is available with constructors.

```
struct constraint_t
{
    constraint_t() {}
    constraint_t(index_t ci, index_t cj, raw_t vij)
        : i(ci), j(cj), value(vij) {}

    index_t i,j;
    raw_t value;
};
```

3.1.2 Operator

operator <

Synopsis: `bool operator <(const constraint_t& a, const constraint_t& b);`

Description: Comparison of two constraints. An arbitrary ordering has been chosen for sorting purposes. See `constraint.h` for the implementation.

3.2 Header file `fed.h`

3.3 Header file `Valuation.h`

3.4 Header file `partition.h`

3.5 Header file `print.h`

3.6 Header file `inline_fed.h`

3.7 Header file `Federation.h`

Chapter 4

Ruby Wrapper

4.1 Module `udbm`

The easiest way to use the module is to start with:
```
require 'udbm'
include UDBM
```
to include the Ruby file udbm.rb and include the Ruby module UDBM that is defined (similar to a namespace for accessing the classes). The examples assume that "include UDBM" has been executed.

matrix

Synopsis: `matrix`

Description: This is a shortcut for `Matrix.new`.

Return: a Matrix.

Fed

Synopsis: `Fed(dim)`

Description: This is the shortcut function to make writing federations more natural. It expects a valid dimension as argument (≥ 1) and a block that results in one Matrix or an array of matrices (or nil in case of an empty federation). The trick is to use the function `matrix`.
Example:
```
Fed(3) # empty federation of dimension 3
Fed(4) {} # empty federation of dimension 4 (variant)
Fed(2) { matrix <=0 <=-2 <=2 <=0 } # one DBM
Fed(2) {[matrix <=0 <=-1 <=1 <=0,matrix <=0 <=-3 <=3 <=0 ]} # two
```
However, constructing federations explicitly is not recommended since it is not what is used in practice. The class methods `zero` and `init` are here for this purpose.

Return: a Fed.

4.1.1 Class UDBM::Constraint

This class includes the module Comparable and has therefor access to the methods declared in this module.

INF

Description: The module defines the constant INF to access the bound infinity.

initialize

Synopsis: `initialize(b, s = false)`

Description: This is the initialization method called by *new*. A constraint is made of a bound and a strictness flag so the expected arguments are the bound (b) and a boolean saying if the constraint is strict or not.
Example:
```
Constraint.new(2,false)
Constraint.new(1,true)
```

Return: The new instance (actually *new* returns it.).

bound

Synopsis: `bound`

Description: This is the attribute reader for the bound of the constraint.

Return: The bound.

strict?

Synopsis: `strict?`

Description: Test if the constraint is strict. This is in fact an attribute reader for the strictness (boolean).

Return: true if the constraint is strict, false otherwise.

bound=

Synopsis: `bound=(b)`

Description: Set the value of the bound of the constraint. The strictness is changed if the argument is infinity (INF).

Return: b.

strict=

Synopsis: `strict=(s)`

Description: Set the strictness of the constraint. A check is done in case the bound is infinity (INF) to keep the constraint consistent. If the constraint is infinity the strictness does not change.

Return: s.

to_s

Synopsis: `to_s`

Description: String representation of a constraint.

Return: a String.

raw

Synopsis: `raw`

Description: This gives access to the encoded value of constraints. Actually it computes it on-the-fly since this class does not store an encoded constraint but is only here as a helper class.

Return: a numeric (Integer in principle).

`<=>`

Synopsis: `<=>(c)`

Description: Comparison function to make the constraint comparable. The ordering corresponds to the ordering of the encoded value of constraints (raw), which is consistent with the natural ordering of inequalities (< 0,≤ 0,< 1,≤ 1,...).

Return: -1, 0, or 1, as a standard `<=>` method is supposed to do.

4.1.2 Class UDBM::Matrix

This class is here only for convenience and testing. It is not necessary for manipulating DBMs. A Matrix is simply a square matrix of constraints and is used to construct DBMs manually or to get constraints individually from DBMs.

initialize

Synopsis: `initialize(*a)`

Description: The initialization method expects constraints as arguments. No argument will initialize an empty matrix.

Return: self.

`<`

Synopsis: `<(b)`

Description: This operator is used as a trick to add constraints to the matrix. It expects a bound as argument and the added constraint is $(< b)$.

Return: self.

`<=`

Synopsis: `<=(b)`

Description: This operator is used as a trick to add constraints to the matrix. It expects a bound as argument and the added constraint is ($\leq b$).

Return: self.

`<<`

Synopsis: `<<(c)`

Description: This operator is similar to `Array::<<` and adds constraints.

Return: self.

dim

Synopsis: `dim`

Description: Compute the dimension corresponding to the current matrix.

Return: $\lfloor\sqrt{size}\rfloor$.

size

Synopsis: `size`

Description: Access to the number of element in the matrix.

Return: a numeric (the size).

to_s

Synopsis: `to_s`

Description: String representation of a matrix.

Return: a String.

inspect

Synopsis: `inspect`

Description: Special formated string representation to be used in *irb*.

Return: a String.

to_a

Synopsis: `to_a`

Description: Access to the internal array that stores the constraints. Notice that if you do something like `m.to_a << Constraint.new(3,true)`, you will add constraints to the matrix.

Return: an Array.

[ ]

Synopsis: `[ ](i,j)`

Description: Access the element of the matrix at (i, j) (a constraint in principle).

Return: Constraint expected.

set

Synopsis: `set(i,j,c)`

Description: Set the constraint (i, j) of in the matrix to be c.

Return: self.

each

Synopsis: `each{ |x| ...}`

Description: Enumerate the constraints of the matrix.

Return: self.

4.1.3 Class UDBM::Relation

The class Relation encapsulates the different constants representing the possible results of a relation between DBMs (or in fact federations here). When applying a relation from a federation on another, e.g., `a.relation(b)`, a Relation result is returned.

Relation::Different

Description: Relation result when the two federations are not comparable.

Relation::Subset

Description: Relation result when the federation 'a' is a subset of 'b'.

Relation::Superset

Description: Relation result when the federation 'a' is a superset of 'b'.

Relation::Equal

Description: Relation result when the federation 'a' is equal to 'b'.

==

Synopsis: `==(r)`

Description: Test equality with another relation.

Return: true or false.

new

Synopsis: `new(i)`

Description: It is possible to create relation instances on-the-fly. In this case the argument is expected to be an integer between 0 and 3 to be mapped to the proper constant.

Return: self.

to_i

Synopsis: `to_i`

Description: Convert a Relation to an Integer corresponding to the internal numerical representation of relations.

Return: a Fixnum.

to_s

Synopsis: `to_s`

Description: Convert a Relation to its String representation.

Return: a String.

4.1.4 Class UDBM::Fed

The class Fed is a wrapper for the underlying C++ fed_t class. There is no support for individual DBMs from Ruby since it is only a particular case of federations. If for some reason the clock constraints need to be accessed, it is possible to get them via the Matrix class.

new

Synopsis: `new(*a)`

Description: The constructor takes a variable number of arguments. The different ways to construct a federation are: (1) Constructor by copy, expects one Fed; and (2) explicit DBMs, expects a list of Matrix instances with the same dimension. There is a shortcut by using the function Fed(dim) with a syntax that corresponds to the output of federations.

Return: a Fed.

Fed.zero

Synopsis: `Fed.zero(dim)`

Description: Create a federation with one DBM of dimension dim representing the origin (the zero point).

Return: a Fed.

Fed.init

Synopsis: `Fed.init(dim)`

Description: Create a federation with one DBM of dimension dim that is unconstrained (infinity for all constraints except the lower bounds set to 0 since clocks are positive).

Return: a Fed.

Fed.random

Synopsis: `Fed.random(dim)`

Description: Create a random federation with a random number of DBMs, all of dimension dim. This may be useful for testing.

Return: a Fed.

initialize

Synopsis: `initialize(*a)`

Description: This is the initialization function called by `new`. See `new`.

Return: a Fed (actually new returns it).

to_s

Synopsis: `to_s`

Description: Standard method to get a String representation of the federation. Notice that the output corresponds to the syntax of declaring federations.

Return: a String.

to_a

Synopsis: `to_a`

Description: Convert a federation to an array of Matrix instances. This is the method to use if you want to access individual constraints for some reason.

Return: an Array.

size

Synopsis: `size`

Description: Return the number of DBMs in the federation.

Return: a Fixnum.

dim

Synopsis: `dim`

Description: Return the dimension of all the DBMs of this federation. Notice that all DBMs must have the same dimension.

Return: a Fixnum.

set_dim!

Synopsis: `set_dim!(dim)`

Description: Empty the federation and change its dimension to dim. As there is no information on which clock constraints to keep, the only way to have a consistent result is to have an empty federation.

Return: self.

copy

Synopsis: `copy`

Description: Return a copy of itself. Keep in mind that doing `a = b` will copy the reference of `b` to `a`, thus modifying `a` (e.g. `a.up!`) will change `b` as well. The copy method gives a new reference.

Return: a new Fed.

empty?

Synopsis: `empty?`

Description: Test if a federation is empty.

Return: true if empty, false otherwise.

unbounded?

Synopsis: `unbounded?`

Description: Test if the federation is unbounded, i.e., if a point in the federation can delay infinitely.

Return: true if unbounded, false otherwise.

empty!

Synopsis: `empty!`

Description: Empty the federation.

Return: self.

intern!

Synopsis: `intern!`

Description: Similarly to Java "intern" call on strings, this call will try to share the DBMs internally so that all instances point to a unique data structure whenever possible, thus saving memory. The user does not have to worry if the federations are modified later, e.g., `a.intern! ...a.up!`, the federations will be consistent and there will be no undesirable side effect.

Return: self.

zero!

Synopsis: `zero!`

Description: Set this federation to the zero point (the origin) with all the clocks equal to zero.

Return: self.

init!

Synopsis: `init!`

Description: Remove all the constraints of this federation (except that all clocks are always positive).

Return: self.

relation

Synopsis: `relation(f)`

Description: Compute a relation in the sense of set inclusion between this federation and the argument federation. The result is typed as a Relation class and is

- Relation::Superset if self is a strict superset of the argument (`self > f`),
- Relation::Subset if self is a strict subset of the argument (`self < f`),
- Relation::Equal if self is equal to the argument (`self == f`), or
- Relation::Different if self is not comparable to the argument.

Return: a Relation.

convex_hull

Synopsis: `convex_hull`

Description: Compute the convex hull of all the DBMs in this federation.

Return: a Fed.

convex_hull!

Synopsis: `convex_hull!`

Description: Set this federation to the convex hull of all its DBMs.

Return: self.

+

Synopsis: `+(f)`

Description: Convex addition of this federation and another. The operator '+' is for convex addition and the '—' for ordinary addition. The result is the convex union of this federation and the argument (with this federation being untouched).

Return: a new Fed.

convex_add!

Synopsis: `convex_add!(f)`

Description: Same a the '+' operator but modify this federation.

Return: self.

constrain_clock!

Synopsis: `constrain_clock!(clock,value)`

Description: Apply the constraint *clock* == *value* for a given clock, which is equivalent to applying both constraints $clock \leq value$ and $clock \geq value$ although it is shorter and faster to call this method.

Return: self.

constrain!

Synopsis: `constrain!(*args)`

Description: Apply a constraint of the form $x_i - x_j < b_{ij}$ or $x_i - x_j \leq b_{ij}$ to this federation. The expected arguments are either i, j, b, s for the indices (i, j), the bound b and a boolean s telling if the bound is strict (true) or not (false), or i, j, c for the indices (i, j) and a Constraint c.

Return: self.

&

Synopsis: `&(f)`

Description: Compute the intersection of this federation and the argument federation.

Return: a new Fed.

intersection!

Synopsis: `intersection!(f)`

Description: Set this federation to the intersection of itself and the argument federation.

Return: self.

intersects?

Synopsis: `intersects?(f)`

Description: Test intersection between this federation and the argument federation. The result is approximate and the "no" answer is safe. To get an exact result (and more expensive to compute), you should use `!(a & b).empty?` which would give the exact wanted result.

Return: true if there *may* be an intersection or false if there is no intersection for sure.

up

Synopsis: `up`

Description: Compute the future. This is the delay, a.k.a. strongest post-condition. In practice this operation removes the upper bounds on the clock constraints.

Return: a new Fed.

up!

Synopsis: `up!`

Description: Apply the delay operation on this federation.

Return: self.

down

Synopsis: `down`

Description: Compute the past. This is the "reverse" delay operation, a.k.a. weakest pre-condition. In practice this operation removes the lower bounds while still maintaining the canonical form.

Return: a new Fed.

down!

Synopsis: `down!`

Description: Apply the past operation on this federation.

Return: self.

free_clock

Synopsis: `free_clock(clock)`

Description: Free all the constraints for a given clock and return a new federation.

Return: a new Fed.

free_clock!

Synopsis: `free_clock!(clock)`

Description: Same as free_clock but modify this federation.

Return: self.

free_up

Synopsis: `free_up(clock)`

Description: Remove the upper bounds of a clock.

Return: a new Fed.

free_up!

Synopsis: `free_up!(clock)`

Description: Same as free_up but modify this federation.

Return: self.

free_down

Synopsis: `free_down(clock)`

Description: Remove the lower bounds of a clock.

Return: a new Fed.

free_down!

Synopsis: `free_down!(clock)`

Description: Same as free_down but modify this federation.

Return: self.

free_all_up

Synopsis: `free_all_up`

Description: Remove all the upper bounds for all the clocks.

Return: a new Fed.

free_all_up!

Synopsis: `free_all_up!`

Description: Same as free_all_up but modify this federation.

Return: self.

free_all_down

Synopsis: `free_all_down`

Description: Remove all the lower bounds for all the clocks.

Return: a new Fed.

free_all_down!

Synopsis: `free_all_down!`

Description: Same as free_all_down but modify this federation.

Return: self.

update_value

Synopsis: `update_value(x,value)`

Description: Apply the update `x := value` for a given clock, where "x" is a clock and "value" a positive integer.

Return: a new Fed.

update_value!

Synopsis: `update_value!(x,value)`

Description: Same as update_value but modify this federation.

Return: self.

update_clock

Synopsis: `update_clock(x,y)`

Description: Apply the update `x := y` for a given clock, where "x" and "y" are clocks.

Return: a new Fed.

update_clock!

Synopsis: `update_clock!(x,y)`

Description: Same as update_clock but modify this federation.

Return: self.

update_increment

Synopsis: `update_increment(x,inc)`

Description: Apply the update `x := x + inc` for a given clock, where "x" is a clock and "inc" is an integer. The user is responsible for making sure that the increment is not too much negative since clocks must stay positive.

Return: a new Fed.

update_increment!

Synopsis: `update_increment!(x,inc)`

Description: Same as update_increment but modify this federation.

Return: self.

update

Synopsis: `update(x,y,value)`

Description: Apply the update `x := y + value` where "x" and "y" are clocks and "value" is an integer. The user is responsible for making sure that the result will give positive clock values.

Return: a new Fed.

update!

Synopsis: `update!(x,y,value)`

Description: Same as update but modify this federation.

Return: self.

satisfies?

Synopsis: `satisfies?(*a)`

Description: Test if this federation satisfies a constraint of the form $x_i - x_j < b_{ij}$ or $x_i - x_j \leq b_{ij}$, i.e., if applying this constraint to the federation results in a non empty federation. Notice that it is fine to test for one constraint but testing several constraints on a row is not correct because constraining a federation with several constraint may yield an empty federation while constraining it with the constraints separately may yield several non empty federations. The arguments may be (i, j, b, s) where (i, j) are the indices, b the bound and s a boolean telling if the constraint is strict ($<$) or not ($\leq$), or (i, j, c) where (i, j) are the indices and c a Constraint.

Return: true if the federation satisfies the constraint, false otherwise.

relax_up

Synopsis: `relax_up`

Description: Make the upper bounds of all the clocks non strict. Notice that some bounds may still be strict if they are inferred by strict diagonal constraints. This method is equivalent to `relax_down_clock(0)`.

Return: a new Fed.

relax_up!

Synopsis: `relax_up!`

Description: Same as relax_up but modify this federation.

Return: self.

relax_down

Synopsis: `relax_down`

Description: Make the lower bounds of all the clocks non strict. Notice that some bounds may still be strict if they are inferred by strict diagonal constraints. This method is equivalent to `relax_up_clock(0)`.

Return: a new Fed.

relax_down!

Synopsis: `relax_down!`

Description: Same as relax_down but modify this federation.

Return: self.

relax_up_clock

Synopsis: `relax_up_clock(clock)`

Description: Make the upper bounds of a particular clock non strict. Some bounds may still be strict if they are inferred by strict diagonal constraints.

Return: a new Fed.

relax_up_clock!

Synopsis: `relax_up_clock!(clock)`

Description: Same as relax_up_clock but modify this federation.

Return: self.

relax_down_clock

Synopsis: `relax_down_clock(clock)`

Description: Make the lower bounds of a particular clock non strict. Some bounds may still be strict if they are inferred by strict diagonal constraints.

Return: a new Fed.

relax_down_clock!

Synopsis: `relax_down_clock!(clock)`

Description: Same as relax_down_clock but modify this federation.

Return: self.

relax_all

Synopsis: `relax_all`

Description: Make all the constraints non strict.

Return: a new Fed.

relax_all!

Synopsis: `relax_all!`

Description: Same as relax_all but modify this federation.

Return: self.

subtraction_empty?

Synopsis: `subtraction_empty?(fed)`

Description: Test if subtraction the argument federation to this federation (self - fed) would give an empty federation, without computing the subtraction itself if possible (internally). This federation is untouched.

Return: `(self-fed).empty?`.

`|`

Synopsis: `|(fed)`

Description: Set union operator between this federation and the argument federation.

Return: a new Fed.

union!

Synopsis: `union!(fed)`

Description: Same as the set union operator (—) but modify this federation.

Return: self.

-

Synopsis: `-(fed)`

Description: Subtraction operator between this federation and the argument federation.

Return: a new Fed.

subtract!

Synopsis: `subtract!(fed)`

Description: Same as the subtraction operator but modify this federation.

Return: self.

merge_reduce!

Synopsis: `merge_reduce!`

Description: Apply a simple reduction algorithm on this federation to merge the DBMs together if possible. The reduction tries to merge DBMs by pairs and is able to remove included DBMs. The method computes a fixpoint internally so there is no point in calling the method several times.

Return: self.

convex_reduce!

Synopsis: `convex_reduce!`

Description: Apply a more complex reduction based on convex union of several DBMs chosen by some heuristic to merge them together if possible. This method behaves at least as well as `merge_reduce` but is more expensive.

Return: self.

partition_reduce!

Synopsis: `partition_reduce!`

Description: Partition the federation and run reduction algorithms on the partitions. This method is more expensive but is as least as good as `convex_reduce`.

Return: self.

expensive_reduce!

Synopsis: `expensive_reduce!`

Description: Apply an expensive reduction algorithm based on subtractions to eliminate included DBMs in the federation. Results may be very different compared to the other reduction algorithms (although `partition_reduce` is using it internally).

Return: self.

expensive_convex_reduce!

Synopsis: `expensive_convex_reduce!`

Description: Recompute the federation, which is potentially very expensive but can give very good results with respect to the number of DBMs.

Return: self.

Important note. All the reduce operations have side-effects in the sense that if several federations refer to the same internal structure, they will all be affected. The set is not changed semantically but it may be simplified. In practice you may notice it even with a copy (which might not update the graphical viewer automatically if you are using it).

predt

Synopsis: `predt(bad)`

Description: Compute the predt operation described in [3]. This federation is considered to represent "good" states and the argument is a set of "bad" states. The method computes the set of predecessors of "good" avoiding the "bad" states, i.e., when delaying from these states they will not intersect the "bad" states and they will end-up in the "good" ones, if possible of course.

Return: a new Fed.

predt!

Synopsis: `predt!(bad)`

Description: Same as predt but modify this federation.

Return: self.

remove_included_in

Synopsis: `remove_included_in(fed)`

Description: Compare the DBMs of this federation and the argument by pairs and return a copy of this federation without the DBMs that are included in those of the argument federation.

Return: a new Fed.

remove_included_in!

Synopsis: `remove_included_in!(fed)`

Description: Same as remove_included_in but modify this federation and do not copy anything.

Return: self.

`<`

Synopsis: `<(fed)`

Description: Set inclusion test: Test if this federation is strictly included in the argument federation. The test involves more than testing DBMs by pairs, it is a set inclusion test on the whole federation as a set. This is equivalent to `self.relation(fed) == Relation::Subset`, although possibly faster.

Return: true if $self \subset fed$ (in the sense of set inclusion), false otherwise.

`>`

Synopsis: `>(fed)`

Description: Set inclusion test similar to `<`: Test if this federation stricly includes the argument federation. This is equivalent to `self.relation(fed) == Relation::Superset`, although possibly faster.

Return: true if $fed \subset self$ (in the sense of set inclusion), false otherwise.

`<=`

Synopsis: `<=(fed)`

Description: Set inclusion test similar to `<`: Test if this federation is included in or equal to the argument federation. This is equivalent to `self.relation(fed) == Relation::Subset || self.relation(fed) == Relation::Equal` that can be rewritten much more efficiently as `(self.relation(fed) & Relation::Subset) != 0`, although possibly even faster.

Return: true if $self \subseteq fed$ (in the sense of set inclusion), false otherwise.

`>=`

Synopsis: `>=(fed)`

Description: Set inclusion test similar to `<`: Test if this federation includes or is equal to the argument federation. This is equivalent to `self.relation(fed) == Relation::Superset || self.relation(fed) == Relation::Equal` that can be rewritten much more efficiently as `(self.relation(fed) & Relation::Superset) != 0`, although possibly even faster.

Return: true if $fed \subseteq self$ (in the sense of set inclusion), false otherwise.

`==`

Synopsis: `==(fed)`

Description: Set inclusion test similar to `<`: Test if this federation is equivalent to the argument federation. This is equivalent to `self.relation(fed) == Relation::Equal`, although possibley faster.

Return: true if $self = fed$ (in the sense of set inclusion), false otherwise.

contains?

Synopsis: `contains?(vec)`

Description: Test if this federation contains a clock valuation (a point in dimension n counting the reference clock 0). The argument (vec) is either an array of integers or an array of floats giving the coordinate of the point in dimension n (and we have of course *self.dim* == n). A clock valuation is included in a federation iff it satisfies all the constraints of one of its DBMs.

Return: true if the point is included in this federation, false otherwise.

possible_back_delay

Synopsis: `possible_back_delay(vec)`

Description: Compute the "almost max" possible delay backward from a point while still staying inside the federation. It is 'almost max' since we want a discrete value, which cannot me the max when we have strict constraints. The precision is 0.5. 0.0 may be returned if the point is too close to a border. The argument is expected to be an array of floats giving the coordinate of the point in the federation.

Return: a Float.

min_delay

Synopsis: `min_delay(point)`

Description: Compute the minimal delay to wait from a point to enter this federation. 0.0 is returned if the point is inside the federation. Infinity is returned if the point cannot enter this federation by delaying. A point is an array of floats giving the coordinate of a point. The dimension must match the dimension of the federation and the first coordinate (reference clock) is expected to be 0.0.

Return:

max_back_delay

Synopsis: `max_back_delay(point)`

Description: Compute the maximal delay to go back in time (back-delay or past) so that the (past-)point is still in this federation. If the point is or is not in the federation or the federation is not a connected set does not matter. 0.0 is returned if there is no such max delay or the max delay is already 0.0.

Return:

delay

Synopsis: `delay(point)`

Description: Compute an interval delay where the first delay is given by `min_delay` and the second delay begin the minimum delay plus the time it is possible to wait and stay in the federation without leaving it *from* the point plus the minimum delay.

Return:

has_zero?

Synopsis: `has_zero?`

Description: Return True if this federation contains the zero point, false otherwise.

Return:

Extrapolation algorithms. The following different extrapolation algorithms are described in [2] and the names correspond to the algorithms of the paper. The arguments to the methods are arrays of integers giving the constants but it is possible to give the special value `-INF`, which has the effect to "free" the corresponding clock, which is a trick for implementing active clock reduction. The dimension of the array argument and the federation should match.

extrapolate_max_bounds

Synopsis: `extrapolate_max_bounds(vec)`

Description: Classical extrapolation using maximal constants.

Return: a new Fed.

extrapolate_max_bounds!

Synopsis: `extrapolate_max_bounds!(vec)`

Description: Same as extrapolate_max_bounds but modify this federation.

Return: self.

diagonal_extrapolate_max_bounds

Synopsis: `diagonal_extrapolate_max_bounds(vec)`

Description: Diagonal extrapolation based on maximal bounds.

Return: a new Fed.

diagonal_extrapolate_max_bounds!

Synopsis: `diagonal_extrapolate_max_bounds!(vec)`

Description: Same as diagonal_extrapolate_max_bounds but modify this federation.

Return: self.

extrapolate_lu_bounds

Synopsis: `extrapolate_lu_bounds(low,up)`

Description: Extrapolation based on lower (low) and upper (up) bounds.

Return: a new Fed.

extrapolate_lu_bounds!

Synopsis: `extrapolate_lu_bounds!(low,up)`

Description: Same as extrapolate_lu_bounds but modify this federation.

Return: self.

diagonal_extrapolate_lu_bounds

Synopsis: `diagonal_extrapolate_lu_bounds(low,up)`

Description: Diagonal extrapolation based on lower (low) and upper (up) bounds.

Return: a new Fed.

diagonal_extrapolate_lu_bounds!

Synopsis: `diagonal_extrapolate_lu_bounds!(low,up)`

Description: Same as diagonal_extrapolate_lu_bounds but modify this federation.

Return: self.

drawing

Synopsis: `drawing(border,width,height,x,y)`

Description: Get drawing information for the drawing module (udbm-gtk). The arguments are:

- border: the border to use (in pixels),
- width: the width of the canvas for drawing (in pixels),
- height: the height of the canvas for drawing (in pixels),
- x,y: the clocks to use for the axis x and y.

Drawing is done in 2-D, which means that the federation is projected on two given dimensions. The result is used internally by udbm-gtk.

Return: an Array with the format [[segments1 , polygon , segments2 , info],. . .] where every DBM gets a sub array. "Segments1" are segments for border lines going to the axis x and y, "polygon" is a set of points to draw a DBM, and "segments2" are additional segments for non strict borders of the polygon. "Info" are tuples of the form [x,y,value] to print a value at (x,y).

point_drawing

Synopsis: `point_drawing(point,x,y)`

Description: Get drawing informatino for drawing a point for the drawing module (udbm-gtk). This method must be called after `drawing`. The arguments are the point (same dimension as the federation being displayed), and the clocks to choose for the projection on 2D. It returns an [[x1,y1,x2,y2],[x1,y1,x2,y2]] of the coordinates of two segments for drawing a cross.

Return:

formula

Synopsis: `formula(names)`

Description: Convert this federation to a readable formula (string of characters). The argument is an Array of String to map clocks j to the names `names[j-1]`. The clock reference (0) has no name and is not part of the mapping.

Return: a String.

change_clocks

Synopsis: `change_clocks(clocks)`

Description: Change the clocks and resize the federation. The new federation has its clock constraints coming from the original ones whenever possible. The argument is an Array giving in order which clocks (indices for the integers, without the clock reference) should be present in the new federation, or `nil` if a new clock is to be inserted (then without constraints on it).

Return: a new Fed.

change_clocks!

Synopsis: `change_clocks!(clocks)`

Description: Same as change_clocks but modify this federation.

Return: self.

`<<`

Synopsis: `<<(fed)`

Description: Operator similar to `Array::<<` to compute a union of this federation with the argument. This is equivalent to `self.union!(fed)`.

Return: self.

dim=

Synopsis: `dim=(d)`

Description: Set a new dimension and empty the federation. This is equivalent to `self.set_dim(d)` but returns d.

Return: d.

4.2 Module `udbm-callback`

This short module is used by `udbm-gtk` to detect changes on federations to automatically update the graphical viewer. Normally users do not need to use this module directly unless they wish to have a change listener feature for federations. This module makes use of Ruby "magic". The following methods are added in the class **Fed**.

method_added

Synopsis: `method_added(id)`

Description: This method is automatically called whenever a method is defined in Fed. The argument is the name of the method newly created. Of course, it is preferrable to include the callback module before declaring methods but there is a work-around to get rid of this limitation.

Return: nil.

register_method

Synopsis: `register_method(name)`

Description: This is the method that does the actual work of registering a method for callback. Only the methods named "something!" are registered.

Return: nil.

add_change_listener

Synopsis: `add_change_listener(proc)`

Description: Add a change listener to call after any previously registered method was executed. The argument is an object that provides a method `call(Fed,String)`, typically a Proc object with these two arguments. When the listener is called, it is passed the instance of Fed triggering the call and the name of the method invoked.

Return: an Array of listeners.

4.3 Module `udbm-sys`

4.3.1 Quick Start

Let's have a mini-tutorial to start. We use `irb` for the interactive Ruby interpreter. Start with including the modules:

```
prompt> irb
irb(main):001:0> require 'udbm-sys'
=> true
irb(main):002:0> require 'udbm-gtk'
=> true
irb(main):003:0> include UDBM
```

```
=> Object
```

Declare a context for our clocks. The context is named "C" and `c` is our ruby reference to it. We can access our clocks as follows:

```
irb(main):004:0> c=Context.create('C',:x,:y,:z)
=> #<UDBM::Context_C {C.x,C.y,C.z}>
irb(main):005:0> c.x
=> #<UDBM::Context::Clock C.x>
irb(main):006:0> c.y
=> #<UDBM::Context::Clock C.y>
irb(main):007:0> c.z
=> #<UDBM::Context::Clock C.z>
```

We can use a Ruby variable "C" to match the context name (optional). Actually, names with capital letters are supposed to be used for constants in Ruby. Let's also declare two sets referred to by `a` and `b`.

```
irb(main):008:0> C=c
=> #<UDBM::Context_C C.x,C.y,C.z>
irb(main):009:0> a=(C.x>C.y) & (C.z<4) & (C.y<3) =>
#<UDBM::Context_C::Set_C (((C.x>C.y) & (C.z<4)) & (C.y<3))>
irb(main):010:0> b=a & ((C.x<2) | (C.x>=3))
=> #<UDBM::Context_C::Set_C ((((C.x>C.y) & (C.z<4)) & (C.y<3)) &
((C.x<2) | (C.x>=3)))>
```

Please be careful with parenthesis to declare sub-expressions otherwise the operator "&" will be applied to clocks and we want it to be applied to sets. These sets are defined naturally with constraints on clocks as you can see. So far they are not evaluated (internal representation, you don't have to worry about it). You can see them with the following commands:

```
irb(main):011:0> a.show('a')
=> #<UDBM::Context_C::Set_C (C.y<3 & C.y-C.x<0 & C.z<4)>
irb(main):012:0> b.show('b')
=> #<UDBM::Context_C::Set_C (3<=C.x & C.y<3 & C.z<4)|(C.x<2 &
C.y-C.x<0 & C.z<4)>
```

Viewing is done by projecting the federation on two dimensions that the user can choose. Notice that invoking the viewer triggers the evaluation of the sets, which explains why they are now displayed slightly differently. You can use most standard operations (see Fed in Subsection 4.1.4) on sets, for example:

```
irb(main):012:0> b.up!
= > #<UDBM::Context_C::Set_C (3<=C.x & C.y-C.x<0 & C.y-C.z<3 &
C.z-C.x<1 & C.z-C.y<4) | (C.x-C.y<2 & C.x-C.z<2 & C.y-C.x<0 &
C.z-C.y<4)>
```

The viewer is updated automatically thanks to the 'udbm-callback' module (loaded automatically by 'udbm-gtk'). Update operations are made extra

user-friendly with operators defined on-the-fly:

```
irb(main):013:0> b.x=1
=> 1
irb(main):014:0> b.up!
=> #<UDBM::Context_C::Set_C (1<=C.x & C.x-C.y<=1 & C.x-C.z<=1
& C.y-C.z<3 & C.z-C.y<4) | (1<=C.x & C.x-C.y<=1 & C.x-C.z<=1 &
C.y-C.z<2 & C.z-C.y<4)>
```

Access to clock `x` in the set b is natural with `b.x`. Notice that the set keeps a reference to its context, you do not have to worry about it. You can also change context to resize federations, which we will describe later. The current federation of our example can be simplified, which we can do with:

```
irb(main):014:0> b.reduce!
=> #<UDBM::Context_C::Set_C (1<=C.x & C.x-C.y<=1 & C.x-C.z<=1 &
C.y-C.z<3 & C.z-C.y<4)>
```

Now that we've seen the basic idea behind sets, let's examine how they work.

4.3.2 Class Fixnum

The standard class Fixnum has some methods added to make the evaluation of formula easier, i.e., avoid to treat particular cases and adopt Ruby philosophy of untyped objects that respond to methods. These methods are used in evaluating expression of the form `x+c` or `x-c` where "x" should refer to a clock (ID) and "c" should be a constant value (an offset).

clock_id

Synopsis: `clock_id`

Description: Return the clock ID associated with this object. Since this object is not a clock, it is associated with the reference clock 0.

Return: 0.

offset

Synopsis: `offset`

Description: The offset associated with this object. Since this object is simply an integer, the offset is itself.

Return: self.

plus_clock

Synopsis: `plus_clock`

Description: This object is not a "+"-expression.

Return: nil.

minus_clock

Synopsis: `minus_clock`

Description: This object is not a "-"-expression.

Return: nil.

4.3.3 Class UDBM::Context

When defining constraint systems, the user needs clocks. A set of clocks is used to define federations and the size of the set corresponds to the dimension of the federations. Clocks are therefor defined within a "context" that groups these clocks to define federations. A context is created with a name (for the context) and a list of clocks (names too). A context is able to generate basic federations (zero, init, random) and provides access to its clock objects that are defined automatically (that's Ruby magic).

name

Synopsis: `name`

Description: The attribute reader for the name of a context.

Return: a String.

clock_names

Synopsis: `clock_names`

Description: The attribute reader for the names of the clocks of a context. The names are prefixed by the name of the context.

Return: an Array of String.

set_class

Synopsis: `set_class`

Description: The attribute reader for the class corresponding to the set of a context. This is used within the module and users do not need this. Classes are also objects, instances of the class `Class`. In Ruby everything is a class.

Return: a Class.

context_id

Synopsis: `context_id`

Description: The attribute reader for the ID of a context. A context has an ID based on the names of its clocks (used for changing context). The ID is just another string constructed used internally. Users do not need to worry about this method.

Return: a Symbol.

short_names

Synopsis: `short_names`

Description: The attribute reader for the short names of the clocks of a context. The difference with `clock_names` is that the names are not prefixed (hence short). The returned type is different, though it does not matter very much, it's for efficiency reasons when changing context.

Return: an Array of Symbol instances.

Context.create

Synopsis: `Context.create(name, *symbols)`

Description: Create a context with a given name and a list of symbols (or strings) for the clocks.
Example:
`c=Context.create(''c'',:x,:y)`
A context has a unique name, which means you cannot declare different contexts with the same name even if they have the same clocks.

Return: a Context.

Context.get

Synopsis: `Context.get(name)`

Description: Get a previously defined context by its name. If the corresponding context was not defined then the methods returns nil.

Return: a Context or nil.

initialize

Synopsis: `initialize(name, *symbols)`

Description: The initialization method of Context called by `new`. Do not create contexts directly with `new`, use `create` instead.

Return: a Context (actually new returns it).

dim

Synopsis: `dim`

Description: The dimension of the federations that can be defined with context. The dimension is equal to the number of clocks plus one (the reference clock with the ID equal to 0, see 4.3.2).

Return: a Fixnum.

zero

Synopsis: `zero`

Description: Create the federation representing the origin (point zero) within this context, i.e., with the clocks defined in this context.

Return: a Fed.

true

Synopsis: `true`

Description: Create an unconstrained federation, i.e., corresponding to true in terms of constraint (or the method `init` on Fed), with the clocks defined in this context.

Return: a Fed.

false

Synopsis: `false`

Description: Create an empty federation, i.e., corresponding to false in terms of constraints, with the clocks defined in this context.

Return: a Fed.

random

Synopsis: `random`

Description: Create a random set, useful for testing only.

Return: a Fed.

to_s

Synopsis: `to_s`

Description: The string representation of this context. This shows a list of clocks.

Return: a String.

update

Synopsis: `update(context)`

Description: Compute the clock mapping from itself to the given context argument for changing context (resizing of federations). The reference clock is always mapped to 0 and is therefor not part of the mapping. The result is an array of integers (IDs of the clocks) of the clocks of this context (self) to be used in a given order for the target context. Newly created clocks have `nil` entries. Example:

```
a=Context.create("A",:x,:y)
```

```
=> #<UDBM::Context_A {A.x,A.y}>
b=Context.create("B",:y,:x,:z)
=> #<UDBM::Context_B {B.y,B.x,B.z}>
a.update(b)
=> [2, 1, nil]
b.update(a)
=> [2, 1]
```
This method is used internally and may be used for other purposes.

Return: an Array of Integer (or nil).

4.3.4 Class UDBM::Context::Clock

The most useful features of clocks are their operators that allow users to write formulas naturally to define constraints. The other methods are useful internally to evaluate formulas to federations.

context

Synopsis: `context`

Description: The attribute reader for the context of this clock.

Return: a Context.

initialize

Synopsis: `initialize(context, index)`

Description: The initialization method to be called by `new`. Users are not supposed to define clocks directly but to access them from a context.

Return: a Clock (actually `new` returns it).

to_s

Synopsis: `to_s`

Description: String representation of this clock, i.e., its name prefixed by the context name.

Return: a String.

clock_id

Synopsis: `clock_id`

Description: The clock ID of this clock, which corresponds to its index in federations (of this clock's context) to access it.

Return: a Fixnum.

offset

Synopsis: `offset`

Description: The offset of this object. Since this is a clock, its offset is 0. Integers have offsets equal to themselves. This is used internally for evaluating formulas.

Return: a Fixnum.

plus_clock

Synopsis: `plus_clock`

Description: The "+ clock" expression of this formula, which is itself for a simple clock. Actually, the method returns the ID of this clock.

Return: a Fixnum.

minus_clock

Synopsis: `minus_clock`

Description: The "- clock" expression of this formula, which is nothing for a simple clock so the method returns nil.

Return: nil.

Operators On Clocks

Clocks have operators in order to write formulas of clock constraints to describe sets. The operators that return sets (Context::Set_XX with XX being the name of the context) are: `==`, `<`, `<=`, `>`, and `>=`. The operators that return sub-formulas (Context::Formula) are `+` and `-`. Notice that the class Formula has the same operators to write formulas.

4.3.5 Class UDBM::Context::Set

The class Set represents sets of clock valuations. This class is not supposed to be created directly by users. Instead, the Context class will create the proper Set_XX class on-demand where XX is the name of the context. Only sets with the same context are compatible. For the sake of simplicity we refer only to Set from now on. Sets are represented internally as formulas (class Formula) or federations (class SymbolicSet that wraps Federation). The convertion is transparent for the user. Sets implement lazy evaluation of formulas, which means that they are evaluated only when needed.

instance

Synopsis: `instance`

Description: Attribute reader for the actual representation of the set (formula or federation). There is also a corresponding attribute writer that should disappear one day.

Return: a Formula or a SymbolicSet.

initialize

Synopsis: `initialize(instance)`

Description: The initialization method called by new. Users do not instantiate sets directly. This is used by Context. The instance is either a Formula or a SymbolicSet that are owned by this Set instance.

Return: a Set (actually `new` returns it).

fed

Synopsis: `fed`

Description: Compute (if needed) the corresponding federation and return it. Since lazy evaluation is used, this is the method to call to get the federation representation.

Return: a Fed.

context

Synopsis: `context`

Description: Get the context of this set.

Return: a Context.

to_s

Synopsis: `to_s`

Description: Convertion to string.

Return: a String.

to_context

Synopsis: `to_context(ctx)`

Description: Change the context of this set. Underlying federations are computed (if needed) and they are resized by removing absent clocks in the target context and added new clocks of the target context. Clocks may also be reordered. This method returns a new set and it is not possible to have a method changing a set without making a copy otherwise the clock_name methods would be incorrect. In addition, the context changes and by design, the Set class too.

Return: a new Set (of different sub-type).

assign_clock!

Synopsis: `assign_clock!(clk, arg)`

Description: Assign the clock *clk* of this set to the argument *arg*. The clock is a Clock instance (see `assign_clock_id!` for using a clock ID). The argument is an integer (Fixnum), another clock (Clock), or a simple formula (Formula) of the form $x + c$ where x is a clock and c an integer (or $x - c$, $c + x$, but not $c - x$).

Return: self.

assign_clock_id!

Synopsis: `assign_clock_id!(clkid, arg)`

Description: Similar to `assign_clock!` but with the clock ID instead of the clock object itself.

Return: self.

copy

Synopsis: `copy`

Description: Copy this set.

Return: a new reference to copy of this Set.

and!

Synopsis: `and!(s)`

Description: Apply the intersection operation to this set. The argument is another set that can be a simple formula, in which case the user gets the effect of constraining the set.

Return: self.

or!

Synopsis: `or!(s)`

Description: Apply the set union operation to this set. The argument is another set (possibly a formula).

Return: self.

subtract!

Synopsis: `subtract!(s)`

Description: Apply the set subtraction operation to this set. The argument is another set (possibly a formula).

Return: self.

`&`

Synopsis: `&(s)`

Description: Intersection operator between two sets.

Return: a new Set.

|

Synopsis: `|(s)`

Description: Union operator between two sets.

Return: a new Set.

-

Synopsis: `-(s)`

Description: Subtraction operator between two sets.

Return: a new Set.

satisfies?

Synopsis: `satisfies?(s)`

Description: Shortcut method for `!(self & s).fed.empty?`. Notice that from the Set class the `satisfies?` method is substantially more expensive than from Fed but on the other hand it is more general and more powerful since it can operate on any formula (i.e., set), including disjunctions of more complex sub-expressions.

Return: true if this set satisfies the constraints given in arguments (represented as a formula or another set), false otherwise.

intern!

Synopsis: `intern!`

Description: This is similar to the `intern!` call from Fed. If the internal representation is a federation then its DBMs are shared between other federations (otherwise nothing happens).

Return: self.

reduce1!

Synopsis: `reduce1!`

Description: The reduce methods are classified subjectively by cost, from cheapest and rather efficient to more expensive and uncertain result. The `reduce1` call corresponds to `Fed::merge_reduce!`.

Return: self.

reduce2!

Synopsis: `reduce2!`

Description: The `reduce2!` call corresponds to `Fed::convex_reduce!`.

Return: self.

reduce3!

Synopsis: `reduce3!`

Description: The `reduce3!` call corresponds to `Fed::expensive_convex_reduce!`.

Return: self.

reduce4!

Synopsis: `reduce4!`

Description: The `reduce4!` call corresponds to `Fed::partition_reduce!`.

Return: self.

reduce5!

Synopsis: `reduce5!`

Description: The `reduce5!` call corresponds to `Fed::expensive_reduce!`.

Return: self.

reduce!

Synopsis: `reduce!`

Description: This is the default reduce method to be called in general without worrying too much about cost. It corresponds to `reduce1!`.

Return: self.

Wrapper Methods (Fed to Set)

Sets have operators and methods that are simple wrappers to their internal federation representation (class Fed). Their semantics are equivalent to their corresponding calls for Fed the difference being the type of argument expected, being sets instead of federations. The readers is referred to Subsection 4.1.4 for the definition of the methods in Fed. The different wrapper methods are:

- Methods taking no argument, returning `true` or `false`:
  empty? and unbounded?.
- Methods taking no argument, returning `self`:
  empty!, convex_hull!, up!, down!, free_all_up!, free_all_down!, relax_up!, relax_down!, relax_all!.
- Methods taking a Set argument, returning `self`:
  remove_included_in!, predt!.
- Methods taking a Set argument, returning a `Relation` (first method), or `true` or `false` to test for relation between sets: relation, `<`, `>`, `<=`, `>=`, `==`.
- Methods taking a Clock argument, returning `self`:
  free_clock!, free_up!, free_down!, relax_up_clock!, relax_down_clock!.

- contains? takes an Array of integers or floats and returns `true` or `false`.
- possible_back_delay takes an Array of floats and returns a float.
- Methods taking an Array of integers, returning `self`:
  extrapolate_max_bounds!, diagonal_extrapolate_max_bounds!.
- Methods taking two Arrays of integers, returning `self`:
  extrapolate_lu_bounds!, diagonal_extrapolate_lu_bounds!.

4.3.6 Internal Classes

Sets are represented internally by either an instance of Formula or SymbolicSet. The convertion of Formula to SymbolicSet is done by evaluating the formula on-the-fly, constraining federations or computing more expensive operations like intersection or unions when needed. We do not describe these classes in detail since users are not supposed to manipulate them directly. We only go through their useful functionalities from a user's point-of-view.

Formulas are represented as simple binary trees and have little semantics, i.e., it is possible to write formulas corresponding to empty sets and manipulate them. When they are converted to SymbolicSet they receive the semantics of federations, which explains why formulas are restricted. It may still be possible to write (buggy) formulas that will fail to convert to federations. Printing formulas corresponds to going through the binary tree and printing the left branch, the operator, and the right branch. Printing SymbolicSet corresponds to computing the minimal graph for every DBM of the underlying federation, print the remaining constraints, and disjunct with all the DBMs.

Operations on sets are actually done on formulas or federations. Since we are implementing lazy evaluation, formulas are converted to federations as late as possible. In particular the simple operations of conjunctions and disjunctions do not trigger the conversion. In addition, user should be aware that consecutive conjunctions are cheaper than mixing conjunctions and disjunctions when the conversion is triggered because they correspond to simple constrain calls. If a sub-expression is a simple conjunction then simple constraining is used (EvalConjunction), otherwise a more general call to compute intersections etc... is used (Eval).

4.4 Module `udbm-gtk`

This short module implements the graphical viewer for federations. It is designed to work with both udbm-sys or udbm, i.e., low-level federations or higher-level sets. Its basic idea is to install a change listener for every method that changes federations (to monitor them) and to redraw them automatically when needed without changing a single line of code in the modules udbm or udbm-sys. The only hook we have currently is for obtaining graphical coordinates since we need to access constraints and compute conversions (scale, project, etc...), which is better to do in C++ since the module is not designed to play with constraints individually. The module adds some hook to other classes to add basic functionalities but we do not modify existing code.

It recommanded to include this module *after* 'udbm-sys' if you are using 'udbm-sys'.

get_fed

Synopsis: `get_fed(title)`

Description: This is the reverse mapping function to get back a federation that is shown under a certain title in the viewer. See Fed::show in Subsection 4.4.4.

Return: a Fed (or nil if there is no corresponding federation).

4.4.1 Class Array

to_color

Synopsis: `to_color`

Description: This is a convenience method to convert arrays of [red, blue, green] values to Gdk::Color.

Return: a Gdk::Color instance corresponding to the specified color.

4.4.2 Class Gdk::Color

darker

Synopsis: `darker`

Description: This is a convenience method to generate a darker color than this instance. By default, $5/6^{th}$ of the intensity is taken.

Return: a new Gdk::Color.

to_s

Synopsis: `to_s`

Description: A more useful string convertion than the default one.

Return: a String.

4.4.3 Class Gtk::Allocation

to_rectangle

Synopsis: `to_rectangle`

Description: This is a convenience method to convert a Gtk::Allocation to a Gdk::Rectangle. For some reason we get the first one although we need the second one for later calls. This can be useful somewhere else.

Return: a Gdk::Rectangle.

4.4.4 Class UDBM::Fed

show

Synopsis: `show(title, labels=nil)`

Description: Hook to display federation. We could have used a function but it is more natural to ask a federation to show itself. The title argument is the name of the tab under which the federation is drawn. There is a one-to-one mapping of the tag and federation, so if later another federation shows itself under the same tag, it replaces the old one in the viewer. Conversely, it is possible to get a federation, given its tag (in the viewer). The labels arguments are useful when using sets because clocks are named and the viewer can then display names instead of clock numbers (this is done automatically internally). The argument is an array of Strings giving the names of the clocks (except the reference clock 0).

Return: self.

hide

Synopsis: `hide`

Description: Hide itself in the viewer (if it was shown).

Return: self.

4.4.5 Class UDBM::Context::Set

show

Synopsis: `show(name)`

Description: Show this set in the viewer. The behaviour is similar to the call from Fed, with the same one-to-one correspondence for federations representing sets (not the sets themselves). The clocks names are prefixed by the context name in the viewer.

Return: self.

show2

Synopsis: `show2(name)`

Description: Similar to show but display short names (not prefixed by the context name).

Return: self.

hide

Synopsis: `hide`

Description: Hide this set in the viewer.

Return: self.

context=

Synopsis: `context=(ctx)`

Description: This is the only hook that modifies existing code. Actually, it is to fix the change of context for a federation since this requires more than just redrawing the federation. The choice of clocks changes as well. This method is not supposed to be used directly.

Return: self.

4.4.6 Internal Classes

Internal classes are not supposed to be used directly so they will not be described in details. We only explain the global structure. The viewer consists of a window (FedWindow) and a panel (FedPanel) where the federation is actually drawn. The panel is responsible for handling the clock choices (drawing the buttons and handling the events) and drawing the federations themselves. We use a slightly modified component from Gtk (to make the labels more readable) for the tabs. The useful feature to know about the drawing is that it is fully automatic and the user has currently no control about it, which means the drawing has to be intelligent. It features

- Automatic color generation with a generated contrast scale that corresponds to perceived human difference between colors (and just numerical), with colors that are supposed to contrast each other.
- Blending of DBMs to see all of them but also each of them :).
- Captions on the axis to read the constraints.
- Automatic scaling (with aspect ratio maintained).

The design for drawing is simple: The drawing area knows its dimensions and the federation to display, so it asks the federation to give it a list of coordinates and labels to display with the proper scaling (sent as arguments, see the method `drawing` in Subsection 4.1.4, page 50). Then it has a simple loop to draw everything.

The class FedWindow handles opening and closing the viewer with one or several windows (the user can drag the tabs out of the window and drag them in again!). It also maintains the mapping label to federation.

4.5 Module `udbm-mdi`

This is the modified module from Ruby/Gtk2. The original component is Gtk::MDI but it has the problem that default label margins are too small to make the tabs easily clickable. We fixed that. See *http://ruby-gnome2.sourceforge.jp/hiki.cgi?MDI* for more information on the original library.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. In *Theoretical Computer Science*, volume 126, pages 183–235, 1994.

[2] Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. In *TACAS'04*, volume 2988 of *LNCS*, pages 312–326. Springer–Verlag, 2004.

[3] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.

[4] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

Index