MiniFrame

P. Baillehache

September 17, 2018

Contents

Introduction

MiniFrame is a C library providing a framework to implement the MiniMax algorithm.

The user can define the system to which the MiniMax algorithm is apply by implementing the set of functions in files `miniframe-model.h`, `miniframe-inline-model.c` and `miniframe-model.c`.

It supports one or several actor(s) and uses a time limit to control MiniMax expansion. MiniFrame uses time prediction to maximise the number of steps computed inside the time limit and minimize the risk of overcoming this time limit.

The user can choose if MiniFrame should try to reuse previously computed worlds or recompute several times the same world if it's reachable through several transitions. If it reuses previously computed worlds MiniFrame provide the percentage of reused worlds at each step. MiniFrame also provide the time unused and the number of computed worlds at each step to allow the user to estimate performances.

A basic example is given to illustrate how to use MiniFrame, as well as the implementation for the game of Oware.

It uses the `PBErr`, `PBMath` and `GSet` libraries.

1 Interface

1.1 miniframe.h

```
// ============ MINIFRAME.H ================

#ifndef MINIFRAME_H
#define MINIFRAME_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "pberr.h"
#include "pbmath.h"
```

```
#include "gset.h"

// ================= Define ==================

// Default time for expansion, in millisecond
#define MF_DEFAULTTIMEEXPANSION 100
// time_ms = clock() / MF_MILLISECTOCLOCKS
#define MF_MILLISECTOCLOCKS (CLOCKS_PER_SEC * 0.001)

// =========== Interface with the model implementation =============

#include "miniframe-model.h"

// ================= Data structure ==================
typedef struct MFWorld MFWorld;
typedef struct MFTransition {
  // User defined transition
  MFModelTransition _transition;
  // Reference to the world to which this action is applied
  MFWorld* _fromWorld;
  // Reference to the reached world through this action
  // if null it means this action has not been computed
  MFWorld* _toWorld;
  // Array of forecasted value of this action from the pov of each actor
  float _values[MF_NBMAXACTOR];
} MFTransition;

typedef struct MFWorld {
  // User defined status of the world
  MFModelStatus _status;
  // Set of transitions reaching this world
  GSet _sources;
  // Array of value of this world from the pov of each actor
  float _values[MF_NBMAXACTOR];
  // Array to memorize the transitions from this world instance
  MFTransition _transitions[MF_NBMAXTRANSITION];
  // Number of transitions from this world
  int _nbTransition;
} MFWorld;

typedef struct MiniFrame {
  // Nb of steps
  unsigned int _nbStep;
  // Current world instance
  MFWorld* _curWorld;
  // All the computed world instances, ordered by their value from the
  // pov of the preempting player at the previous step
  GSet _worlds;
  // Time limit for expansion, in millisecond
  float _maxTimeExpansion;
  // Time unused during expansion, in millisecond
  float _timeUnusedExpansion;
  // Percent of the total available time available to search for worlds
  // to expand in MFExpand(), in ]0.0, 1.0], init to 1.0
  float _timeSearchWorld;
  // Nb of worlds expanded during last call to MFExpand
  int _nbWorldExpanded;
  // Flag to activate the reuse of previously computed same world
  bool _reuseWorld;
  // Percentage (in [0.0, 1.0]) of world reused during the last
  // MFExpand()
  float _percWorldReused;
```

```
  // Time used at end of expansion (per remaining world)
  float _timeEndExpansion;
  // The clock considered has start during expansion
  clock_t _startExpandClock;
} MiniFrame;


// ================ Functions declaration ====================

// Create a new MiniFrame the initial world 'initStatus'
// The current world is initialized with a copy of 'initStatus'
// Return the new MiniFrame
MiniFrame* MiniFrameCreate(const MFModelStatus* const initStatus);

// Create a new MFWorld with a copy of the MFModelStatus 'status'
// Return the new MFWorld
MFWorld* MFWorldCreate(const MFModelStatus* const status);

// Create a new static MFTransition for the MFWorld 'world' with the
// MFModelTransition 'transition'
// Return the new MFTransition
MFTransition MFTransitionCreateStatic(const MFWorld* const world,
  const MFModelTransition* const transition);

// Free memory used by the MiniFrame 'that'
void MiniFrameFree(MiniFrame** that);

// Free memory used by the MFWorld 'that'
void MFWorldFree(MFWorld** that);

// Free memory used by properties of the MFTransition 'that'
void MFTransitionFreeStatic(MFTransition* that);

// Get the current MFWorld of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFCurWorld(const MiniFrame* const that);

// Get the GSet of computed MFWorlds of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorlds(const MiniFrame* const that);

// Return the number of computed worlds in the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbComputedWorld(const MiniFrame* const that);

// Return true if the expansion algorithm looks in previously
// computed worlds for same world to reuse, else false
#if BUILDMODE != 0
inline
#endif
bool MFIsWorldReusable(const MiniFrame* const that);

// Set the falg controlling if the expansion algorithm looks in
// previously computed worlds for same world to reuse to 'reuse'
#if BUILDMODE != 0
inline
```

```
#endif
void MFSetWorldReusable(MiniFrame* const that, const bool reuse);

// Add the MFWorld 'world' to the computed MFWorlds of the
// MiniFrame 'that', ordered by the world's value from the pov of
// actor 'iActor'
#if BUILDMODE != 0
inline
#endif
void MFAddWorld(MiniFrame* const that, \
  const MFWorld* const world, const int iActor);

// Get the time limit for expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetMaxTimeExpansion(const MiniFrame* const that);

// Get the time unused during last expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeUnusedExpansion(const MiniFrame* const that);

// Get the time used to search world to expand during next expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeSearchWorld(const MiniFrame* const that);

// Get the nb of world expanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldExpanded(const MiniFrame* const that);

// Get the time used at end of expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeEndExpansion(const MiniFrame* const that);

// Get the percentage of resued world of the MiniFrame 'that' during
// the last MFEpxand()
#if BUILDMODE != 0
inline
#endif
float MFGetPercWordReused(const MiniFrame* const that);

// Get the clock considered has start during expansion
#if BUILDMODE != 0
inline
#endif
clock_t MFGetStartExpandClock(const MiniFrame* const that);

// Set the clock considered has start during expansion to 'c'
#if BUILDMODE != 0
inline
#endif
void MFSetStartExpandClock(MiniFrame* const that, clock_t c);
```

```
// Set the time limit for expansion of the MiniFrame 'that' to
// 'timeLimit', in millisecond
// The time is measured with the function clock(), see "man clock"
// for details
#if BUILDMODE != 0
inline
#endif
void MFSetMaxTimeExpansion(MiniFrame* const that, \
  const float timeLimit);

// Return the MFModelStatus of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFModelStatus* MFWorldStatus(const MFWorld* const that);

// Expand the MiniFrame 'that' until it reaches its time limit or can't
// expand anymore
void MFExpand(MiniFrame* that);

// Return the value of the MFWorld 'that' from the point of view of the
// actor 'hero'.
// If 'hero' equals -1 it mens we are getting the value of a world
// during expansion and there is no preempting actor (everyone acting
// at the same time). Then we give priority during expansion to
// worlds maximising simultaneously individual values.
float MFWorldGetPOVValue(const MFWorld* const that, const int hero);

// Return the value of the MFTransition 'that' from the point of view
// of the actor 'hero'.
// If 'hero' equals -1 it means we are getting the value of a world
// during expansion and there is no preempting actor (everyone acting
// at the same time). Then we give priority during expansion to
// worlds maximising simultaneously individual values.
float MFTransitionGetPOVValue(const MFTransition* const that,
  const int hero);

// Get the number of transition for the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
int MFWorldGetNbTrans(const MFWorld* const that);

// Get the MFWorld which the MFTransition 'that' is leading to
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionToWorld(const MFTransition* const that);

// Set the MFWorld to which the MFTransition 'that' is leading to
// 'world'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetToWorld(MFTransition* const that,
  MFWorld* const world);

// Get the MFWorld which the MFTransition 'that' is coming from
#if BUILDMODE != 0
inline
#endif
```

```
const MFWorld* MFTransitionFromWorld(const MFTransition* const that);

// Return true if the MFTransition 'that' is expandable, i.e. its
// 'toWorld' is null, else return false
#if BUILDMODE != 0
inline
#endif
bool MFTransitionIsExpandable(const MFTransition* const that);

// Get the 'iTrans' MFTransition of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFTransition* MFWorldTransition(const MFWorld* const that,
  const int iTrans);

// Get the set of MFTransition reaching the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorldSources(const MFWorld* const that);

// Return the array of values of the MFWorld 'that' for each actor
#if BUILDMODE != 0
inline
#endif
const float* MFWorldValues(const MFWorld* const that);

// Compute the MFModelStatus resulting from the 'iTrans' MFTransition
// of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFModelStatus MFWorldComputeTransition(const MFWorld* const that,
  const int iTrans);

// Get the forecast value of the MFWorld 'that' for the actor 'iActor'
// It's the value of the MFWorldif it has no transitions, or the
// highest value of its transitions
float MFWorldGetForecastValue(const MFWorld* const that, int iActor);

// Set the value of the MFTransition 'that' for the actor 'iActor' to
// 'val'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetValue(MFTransition* const that, const int iActor,
  const float val);

// Return the egocentric value of the MFTransition 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFTransitionGetValue(const MFTransition* const that,
  const int iActor);

// Return the egocentric value of the MFWorld 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
```

```
float MFWorldGetValue(const MFWorld* const that, const int iActor);

// Get the best MFModelTransition for the 'iActor'-th actor in the
// current MFWorld of the MiniFrame 'that'
// Return an undefined MFTransition if the curernt world has no
// transition
const MFModelTransition* MFGetBestTransition(
  const MiniFrame* const that, const int iActor);

// Print the MFWorld 'that' on the stream 'stream'
void MFWorldPrint(const MFWorld* const that, FILE* const stream);

// Print the MFTransition 'that' on the stream 'stream'
void MFTransitionPrint(const MFTransition* const that,
  FILE* const stream);

// Print the MFWorld 'that' and its MFTransition on the stream 'stream'
void MFWorldTransPrintln(const MFWorld* const that,
  FILE* const stream);

// Set the current world of the MiniFrame 'that' to match the
// MFModelStatus 'world'
void MFSetCurWorld(MiniFrame* const that,
  const MFModelStatus* const world);

// ================ Inliner ====================

#if BUILDMODE != 0
#include "miniframe-inline.c"
#endif


#endif
```

2 Code

2.1 miniframe.c

```
// ============ MINIFRAME.C ================

// ================= Include =================

#include "miniframe.h"
#if BUILDMODE == 0
#include "miniframe-inline.c"
#endif

// ================ Functions declaration ====================

// Get the set of worlds to be expanded (having at least one transition
// whose _toWorld is null) for the MiniFrame 'that'
// Stop searching for world if clock() >= clockLimit
// Will return at least one world even if clockLimit == current clock
// The MiniFrame must have at least one world in its set of computed
// worlds
// Force the current world to the end of the returned set to ensure
// it will be the first to be expanded
GSet MFGetWorldsToExpand(const MiniFrame* const that,
```

```
  const clock_t clockLimit);

// Return true if the MFWorld 'that' has at least one transition to be
// expanded
bool MFWorldIsExpandable(const MFWorld* const that);

// Search in computed worlds of the MiniFrame 'that' if there is
// one with same status as the MFModelStatus 'status'
// If there is one return it, if not return null
MFWorld* MFSearchWorld(const MiniFrame* const that,
  const MFModelStatus* const status);

// Set the MFWorld 'toWorld' has the result of the 'iTrans' transition
// of the world 'that'
// Update the value of the transition
void MFWorldSetTransitionToWorld(
  MFWorld* const that, const int iTrans, MFWorld* const toWorld);

// Update backward the forecast values for each transitions
// leading to the MFWorld 'world' in the MiniFrame 'that'
void MFUpdateForecastValues(MiniFrame* const that,
  const MFWorld* const world, float delayPenalty);

// Update the values of the MFTransition 'that' for each actor with
// the values 'values'
// Update only if the new value is higher than the current one
// Return true if at least one value has been updated, else false
bool MFTransitionUpdateValues(MFTransition* const that,
  const float* const values);

// Pop a MFTransition from the sources of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFTransition* MFWorldPopSource(MFWorld* const that);

// Remove the MFTransition 'source' from the sources of the
// MFWorld 'that'
void MFWorldRemoveSource(MFWorld* const that,
  const MFTransition* const source);

// ================ Functions implementation ====================

// Create a new MiniFrame the initial world 'initStatus'
// The current world is initialized with a copy of 'initStatus'
// Return the new MiniFrame
MiniFrame* MiniFrameCreate(const MFModelStatus* const initStatus) {
#if BUILDMODE == 0
  if (initStatus == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'initStatus' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Allocate memory
  MiniFrame *that = PBErrMalloc(MiniFrameErr, sizeof(MiniFrame));
  // Set properties
  that->_nbStep = 0;
  MFSetMaxTimeExpansion(that, MF_DEFAULTTIMEEXPANSION);
  that->_curWorld = MFWorldCreate(initStatus);
  that->_worlds = GSetCreateStatic();
  MFAddWorld(that, MFCurWorld(that), -1);
```

```
  that->_timeSearchWorld = MF_DEFAULTTIMEEXPANSION;
  that->_nbWorldExpanded = 0;
  that->_timeUnusedExpansion = 0.0;
  that->_reuseWorld = false;
  that->_percWorldReused = 0.0;
  that->_startExpandClock = 0;
  // Estimate the time used at end of expansion which is the time
  // used to flush a gset
  GSet set = GSetCreateStatic();
  int nb = 100;
  float timeFlush = 0.0;
  do {
    for (int i = nb; i--;)
      GSetPush(&set, NULL);
    clock_t timeStart = clock();
    GSetFlush(&set);
    clock_t timeEnd = clock();
    timeFlush = ((double)(timeEnd - timeStart)) / MF_MILLISECTOCLOCKS;
  } while (timeFlush < 0.0);
  that->_timeEndExpansion = timeFlush / (float)nb;
  // Return the new MiniFrame
  return that;
}

// Create a new MFWorld with a copy of the MFModelStatus 'status'
// Return the new MFWorld
MFWorld* MFWorldCreate(const MFModelStatus* const status) {
#if BUILDMODE == 0
  if (status == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'status' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Allocate memory
  MFWorld *that = PBErrMalloc(MiniFrameErr, sizeof(MFWorld));
  // Set the status
  MFModelStatusCopy(status, &(that->_status));
  // Initialise the set of transitions reaching this world
  that->_sources = GSetCreateStatic();
  // Set the values
  for (int iActor = MF_NBMAXACTOR; iActor--;)
    that->_values[iActor] = 0.0;
  MFModelStatusGetValues(status, that->_values);
  // Set the possible transitions from this world
  MFModelTransition transitions[MF_NBMAXTRANSITION];
  MFModelStatusGetTrans(status, transitions, &(that->_nbTransition));
  for (int iTrans = that->_nbTransition; iTrans--;)
    that->_transitions[iTrans] =
      MFTransitionCreateStatic(that, transitions + iTrans);
  // Return the new MFWorld
  return that;
}

// Create a new static MFTransition for the MFWorld 'world' with the
// MFModelTransition 'transition'
// Return the new MFTransition
MFTransition MFTransitionCreateStatic(const MFWorld* const world,
  const MFModelTransition* const transition) {
#if BUILDMODE == 0
  if (world == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(MiniFrameErr->_msg, "'world' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the new action
  MFTransition that;
  // Set properties
  that._transition = *transition;
  that._fromWorld = (MFWorld*)world;
  that._toWorld = NULL;
  for (int iActor = MF_NBMAXACTOR; iActor--;)
    that._values[iActor] = 0.0;
  // Return the new MFTransition
  return that;
}

// Free memory used by the MiniFrame 'that'
void MiniFrameFree(MiniFrame** that) {
  // Check argument
  if (that == NULL || *that == NULL) return;
  // Free memory
  while(GSetNbElem(&((*that)->_worlds)) > 0) {
    MFWorld* world = GSetPop(&((*that)->_worlds));
    MFWorldFree(&world);
  }
  free(*that);
  *that = NULL;
}

// Free memory used by the MFWorld 'that'
void MFWorldFree(MFWorld** that) {
  // Check argument
  if (that == NULL || *that == NULL) return;
  // Free memory
  GSetFlush(&((*that)->_sources));
  MFModelStatusFreeStatic(&((*that)->_status));
  for (int iAct = (*that)->_nbTransition; iAct--;) {
    if ((*that)->_transitions[iAct]._toWorld != NULL)
      MFTransitionFreeStatic((*that)->_transitions + iAct);
  }
  free(*that);
  *that = NULL;
}

// Free memory used by properties of the MFTransition 'that'
void MFTransitionFreeStatic(MFTransition* that) {
  // Check argument
  if (that == NULL) return;
  // Free memory
  MFModelTransitionFreeStatic(&(that->_transition));
}

// Expand the MiniFrame 'that' until it reaches its time limit or can't
// expand anymore
void MFExpand(MiniFrame* that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
```

```
// Declare a variable to memorize the time at beginning of the whole
// expansion process
clock_t clockStart = MFGetStartExpandClock(that);
// Declare a variable to memorize the maximum time used for one
// step of expansion
double maxTimeOneStep = 0.0;
// Create the set of world instances to be expanded, ordered by
// world's value from the point of view of the preempting actor
// for each world
// The time available for this step is limited to avoid spending
// time to search for worlds to expand and finally not having time
// to compute them
clock_t clockLimit = clockStart +
  that->_timeSearchWorld * MF_MILLISECTOCLOCKS;
GSet worldsToExpand = MFGetWorldsToExpand(that, clockLimit);
// Memorize the number of worlds to expand
int nbWorldToExpand = GSetNbElem(&worldsToExpand);
// Declare a variable to memorize the time spend expanding
double timeUsed =
  ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
// Declare a variable to memorize the number of reused worlds
int nbReusedWorld = 0;
// Loop until we have time for one more step of expansion or there
// is no world to expand
// Take care of clock() wrapping around
while (timeUsed >= 0.0 &&
  timeUsed + maxTimeOneStep +
  MFGetTimeEndExpansion(that) * GSetNbElem(&worldsToExpand) <
  MFGetMaxTimeExpansion(that) &&
  GSetNbElem(&worldsToExpand) > 0) {
  // Declare a variable to memorize the time at the beginning of one
  // step of expansion
  clock_t clockStartLoop = clock();
  // Drop the world to expand with highest value
  MFWorld* worldToExpand = GSetDrop(&worldsToExpand);
  // Get the sente for this world
  int sente = MFModelStatusGetSente(MFWorldStatus(worldToExpand));
  // For each transitions from the expanded world and until we have
  // time available
  // Take care of clock() wrapping around
  for (int iTrans = 0; iTrans < MFWorldGetNbTrans(worldToExpand) &&
    timeUsed >= 0.0 &&
    timeUsed + maxTimeOneStep < MFGetMaxTimeExpansion(that);
    ++iTrans) {
    // If this transition has not been computed
    const MFTransition* const trans =
      MFWorldTransition(worldToExpand, iTrans);
    if (MFTransitionIsExpandable(trans)) {
      // Expand through this transition
      MFModelStatus status =
        MFWorldComputeTransition(worldToExpand, iTrans);
      // If the resulting status has not already been computed
      MFWorld* sameWorld = MFSearchWorld(that, &status);
      if (sameWorld == NULL) {
        // Create a MFWorld for the new status
        MFWorld* expandedWorld = MFWorldCreate(&status);
        // Add the world to the set of computed world
        MFAddWorld(that, expandedWorld, sente);
        // Add the world to the set of worlds to expand
        ++nbWorldToExpand;
        int sente =
          MFModelStatusGetSente(MFWorldStatus(expandedWorld));
```

```
        float value = MFWorldGetPOVValue(expandedWorld, sente);
        GSetAddSort(&worldsToExpand, expandedWorld, value);
        // Set the expanded world as the result of the transition
        MFWorldSetTransitionToWorld(
          worldToExpand, iTrans, expandedWorld);
      } else {
        // Increment the number of reused world
        ++nbReusedWorld;
        // Set the already computed one as the result of the
        // transition
        MFWorldSetTransitionToWorld(worldToExpand, iTrans, sameWorld);
      }
    }
    // Update the total time used from beginning of expansion
    timeUsed =
      ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
  }
  // Update backward the forecast values for each transitions
  // leading to the expanded world according to its new transitions
  MFUpdateForecastValues(that, worldToExpand, 0.0);
  // Declare a variable to memorize the time at the end of one
  // step of expansion
  clock_t clockEndLoop = clock();
  // Calculate the time for this step
  double timeOneStep =
    ((double)(clockEndLoop - clockStartLoop)) / MF_MILLISECTOCLOCKS;
  // Update max time used by one step
  if (maxTimeOneStep < timeOneStep)
    maxTimeOneStep = timeOneStep;
  // Update the total time used from beginning of expansion
  timeUsed =
    ((double)(clockEndLoop - clockStart)) / MF_MILLISECTOCLOCKS;
}
// Memorize the remaining number of worlds to expand
int nbRemainingWorldToExpand = GSetNbElem(&worldsToExpand);
// Update the total time used from beginning of expansion
timeUsed = ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
// Update the percentage of time allocated to searching for worlds
// to expand
// If we could expand all the worlds
if (nbRemainingWorldToExpand == 0) {
  if (timeUsed > PBMATH_EPSILON) {
    that->_timeSearchWorld *=
      MFGetMaxTimeExpansion(that) / timeUsed;
    if (that->_timeSearchWorld > MFGetMaxTimeExpansion(that))
      that->_timeSearchWorld = MFGetMaxTimeExpansion(that);
  } else {
    that->_timeSearchWorld = MFGetMaxTimeExpansion(that);
  }
// Else, we had not enough time to expand all the worlds
} else {
  that->_timeSearchWorld *=
    (float)nbRemainingWorldToExpand / (float)nbWorldToExpand;
}
// Empty the list of worlds to expand
GSetFlush(&worldsToExpand);
// Update the total time used from beginning of expansion
timeUsed = ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
// Take care of clock() wrapping around
if (timeUsed < 0.0)
  timeUsed = MFGetMaxTimeExpansion(that);
// Telemetry for debugging
```

```
  that->_timeUnusedExpansion = MFGetMaxTimeExpansion(that) - timeUsed;
  that->_nbWorldExpanded =
    nbWorldToExpand - nbRemainingWorldToExpand + nbReusedWorld;
  if (that->_nbWorldExpanded > 0)
    that->_percWorldReused =
      (float)nbReusedWorld / (float)(that->_nbWorldExpanded);
  else
    that->_percWorldReused = 0.0;
}

// Return the value of the MFWorld 'that' from the point of view of the
// actor 'hero'.
// If 'hero' equals -1 it mens we are getting the value of a world
// during expansion and there is no preempting actor (everyone acting
// at the same time). Then we give priority during expansion to
// worlds maximising simultaneously individual values.
float MFWorldGetPOVValue(const MFWorld* const that, const int hero) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (hero < -1 || hero >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'hero' is invalid (-1<=%d<%d)", \
      hero, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the returned value
  float value = 0.0;
  // Loop on actors
  for (int iActor = MF_NBMAXACTOR; iActor--;) {
    // If this actor is active
    if (MFModelStatusIsActorActive(MFWorldStatus(that), iActor)) {
      // Update the value
      if (iActor == hero || hero == -1)
        value += MFWorldGetValue(that, iActor);
      else
        value -= MFWorldGetValue(that, iActor);
    }
  }
  // Return the value
  return value;
}

// Return the value of the MFTransition 'that' from the point of view
// of the actor 'hero'.
// If 'hero' equals -1 it mens we are getting the value of a world
// during expansion and there is no preempting actor (everyone acting
// at the same time). Then we give priority during expansion to
// worlds maximising simultaneously individual values.
float MFTransitionGetPOVValue(const MFTransition* const that,
  const int hero) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (hero < -1 || hero >= MF_NBMAXACTOR) {
```

```
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'hero' is invalid (-1<=%d<%d)", \
      hero, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the returned value
  float value = 0.0;
  // Loop on actors
  for (int iActor = MF_NBMAXACTOR; iActor--;) {
    // If this actor is active
    if (MFModelStatusIsActorActive(
      MFWorldStatus(MFTransitionToWorld(that)), iActor)) {
      // Update the value
      if (iActor == hero || hero == -1)
        value += MFTransitionGetValue(that, iActor);
      else
        value -= MFTransitionGetValue(that, iActor);
    }
  }
  // Return the value
  return value;
}

// Get the set of worlds to be expanded (having at least one transition
// whose _toWorld is null) for the MiniFrame 'that'
// Stop searching for world if clock() >= clockLimit
// Will return at least one world even if clockLimit == current clock
// The MiniFrame must have at least one world in its set of computed
// worlds
// Force the current world to the end of the returned set to ensure
// it will be the first to be expanded
GSet MFGetWorldsToExpand(const MiniFrame* const that,
  const clock_t clockLimit) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (GSetNbElem(MFWorlds(that)) == 0) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "The MiniFrame has no computed world");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare the set to memorize worlds to expand
  GSet set = GSetCreateStatic();
  // Loop through the computed worlds
  GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
  do {
    MFWorld* world = GSetIterGet(&iter);
    // If this world has transition to expand
    if (world != MFCurWorld(that) && MFWorldIsExpandable(world)) {
      // Add this world to the result set ordered by the value
      int sente = MFModelStatusGetSente(MFWorldStatus(world));
      float value = MFWorldGetPOVValue(world, sente);
      GSetAddSort(&set, world, value);
    }
  } while (GSetIterStep(&iter) && clock() < clockLimit);
  // Add the current world
  if (MFWorldIsExpandable(MFCurWorld(that))) {
```

```
    GSetAppend(&set, MFCurWorld(that));
  }
  // Return the set of worlds to expand
  return set;
}

// Return true if the MFWorld 'that' has at least one transition to be
// expanded
bool MFWorldIsExpandable(const MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the result
  bool isExpandable = false;
  // If the world is not at the end of the game/simutation
  if (!MFModelStatusIsEnd(MFWorldStatus(that))) {
    // Loop on transitions
    for (int iTrans = that->_nbTransition; iTrans-- && !isExpandable;) {
      // If this transition has not been computed
      if (MFTransitionIsExpandable(
        MFWorldTransition(that, iTrans)))
        isExpandable = true;
    }
  }
  // Return the result
  return isExpandable;
}

// Search in computed worlds of the MiniFrame 'that' if there is
// one with same status as the MFModelStatus 'status'
// If there is one return it, if not return null
MFWorld* MFSearchWorld(const MiniFrame* const that,
  const MFModelStatus* const status) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (status == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'status' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the returned world
  MFWorld* sameWorld = NULL;
  // If the reuse of worlds is activated
  if (MFIsWorldReusable(that)) {
    // Loop on computed worlds
    GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
    do {
      MFWorld* world = GSetIterGet(&iter);
      // If this world is the same as the searched one
      if (MFModelStatusIsSame(status, MFWorldStatus(world))) {
        sameWorld = world;
      }
    } while (sameWorld == NULL && GSetIterStep(&iter));
```

```
  }
  // Return the found world
  return sameWorld;
}

// Set the MFWorld 'toWorld' has the result of the 'iTrans' transition
// of the MFWorld 'that'
// Update the value of the transition
void MFWorldSetTransitionToWorld(
  MFWorld* const that, const int iTrans, MFWorld* const toWorld) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iTrans < 0 || iTrans >= that->_nbTransition) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
      iTrans, that->_nbTransition);
    PBErrCatch(MiniFrameErr);
  }
  if (toWorld == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'toWorld' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the transition
  MFTransition* trans = that->_transitions + iTrans;
  // Set the transition result
  trans->_toWorld = toWorld;
  // Add the transition to the sources to the result's world
  GSetAppend(&(toWorld->_sources), trans);
  // Update the forecast value of this transition for each actor
  for (int iActor = MF_NBMAXACTOR; iActor--;)
    MFTransitionSetValue(trans, iActor,
      MFWorldGetValue(toWorld, iActor));
}

// Get the forecast value of the MFWorld 'that' for the actor 'iActor'
// It's the value of the MFWorldif it has no transitions, or the
// highest value of its transitions
float MFWorldGetForecastValue(const MFWorld* const that, int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)",
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize if there are transitions
  bool flagTransition = false;
  // Declare a variable to memorize the highest value among transitions
  float valBestTrans = 0.0;
  // Loop on transitions
```

```
  for (int iTrans = MFWorldGetNbTrans(that); iTrans--;) {
    // Declare a variable to memorize the transition
    const MFTransition* const trans =
      MFWorldTransition(that, iTrans);
    // If this transitions has been expanded
    if (!MFTransitionIsExpandable(trans)) {
      // If it's not the first considered transition
      if (flagTransition) {
        // Get the value of the transition from the point of view of
        // the requested actor
        float val =
          MFTransitionGetPOVValue(trans, iActor);
        // If the value is better
        if (valBestTrans < val)
          valBestTrans = val;
      // Else it's the first considered transition
      } else {
        // Init the best value with the value of this transition
        valBestTrans =
          MFTransitionGetPOVValue(trans, iActor);
        // Set the flag to memorize there are transitions
        flagTransition = true;
      }
    }
  }
  // Return the value for this world
  // If there are transitions
  if (flagTransition) {
    // Return the value of the best transition from the point of view
    // of the requested actor
    return valBestTrans;
  // Else this world has no transitions
  } else {
    // Return the value of this world from the point of view of the
    // requested actor
    return MFWorldGetPOVValue(that, iActor);
  }
}

// Get the best MFModelTransition for the 'iActor'-th actor in the
// current MFWorld of the MiniFrame 'that'
// Return an undefined MFTransition if the curernt world has no
// transition
const MFModelTransition* MFGetBestTransition(
  const MiniFrame* const that, const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)",
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the highest value among transitions
  float valBestTrans = 0.0;
  // Declare a variable to memorize the best transition
  const MFTransition* bestTrans = NULL;
```

```
  // Get the current world
  const MFWorld* const curWorld = MFCurWorld(that);
  // Loop on transitions
  for (int iTrans = MFWorldGetNbTrans(curWorld); iTrans--;) {
    // Declare a variable to memorize the transition
    const MFTransition* const trans =
      MFWorldTransition(curWorld, iTrans);
    // If this transitions has been expanded
    if (!MFTransitionIsExpandable(trans)) {
      // If it's not the first considered transition
      if (bestTrans != NULL) {
        // Get the value of the transition from the point of view of
        // the requested actor
        float val = MFTransitionGetPOVValue(trans, iActor);
        // If the value is better
        if (valBestTrans < val) {
          // Update the best value and best transition
          valBestTrans = val;
          bestTrans = trans;
        }
      // Else it's the first considered transition
      } else {
        // Init the best value with the value of this transition
        valBestTrans = MFTransitionGetPOVValue(trans, iActor);
        // Init the best transition
        bestTrans = trans;
      }
    }
  }
  // If the bestTrans is null here it means that none of the transitions
  // for the current world were expanded yet
  // By default choose a random one
  if (bestTrans == NULL && MFWorldGetNbTrans(curWorld) > 0) {
    bestTrans = MFWorldTransition(curWorld,
      (int)floor(MIN(rnd(), 0.9999) * (float)MFWorldGetNbTrans(curWorld)));
  }
  // Return the best transition
  return (const MFModelTransition*)bestTrans;
}

// Update backward the forecast values for each transitions
// leading to the MFWorld 'world' in the MiniFrame 'that'
// Use a penalty growing with each recursive call to
// MFUpdateForecastValues to give priority to fastest convergence to
// best solution and avoid infinite loop due to reuse of computed worlds
void MFUpdateForecastValues(MiniFrame* const that,
  const MFWorld* const world, float delayPenalty) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (world == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'world' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Increase the penalty
  delayPenalty += PBMATH_EPSILON;
  // If the world has ancestors
```

```
  if (GSetNbElem(MFWorldSources(world)) > 0) {
    // Get the forecast values of the world for each actor
    float values[MF_NBMAXACTOR] = {0.0};
    for (int iAct = MF_NBMAXACTOR; iAct--;)
      values[iAct] =
        MFWorldGetForecastValue(world, iAct) - delayPenalty;
    // For each source to the world
    GSetIterForward iter =
      GSetIterForwardCreateStatic(MFWorldSources(world));
    do {
      // Get the transition
      MFTransition* const trans = GSetIterGet(&iter);
      // Update the transition's forecast value
      bool updated = MFTransitionUpdateValues(trans, values);
      // If the transition has been updated
      if (updated) {
        // Call recursively the MFUpdateForecastValues with the origin
        // of this transition
        MFUpdateForecastValues(that, MFTransitionFromWorld(trans),
          delayPenalty + PBMATH_EPSILON);
      }
    } while(GSetIterStep(&iter));
  }
}

// Update the values of the MFTransition 'that' for each actor with
// the values 'values'
// Update only if the new value is higher than the current one
// Return true if at least one value has been updated, else false
bool MFTransitionUpdateValues(MFTransition* const that,
  const float* const values) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (values == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'values' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the returned flag
  bool updated = false;
  // For each actor
  for (int iAct = MF_NBMAXACTOR; iAct--;) {
    if (that->_values[iAct] < values[iAct]) {
      updated = true;
      that->_values[iAct] = values[iAct];
//printf("update ");MFTransitionPrint(that, stdout);printf("\n");
    }
  }
  // Return the flag
  return updated;
}

// Print the MFWorld 'that' on the stream 'stream'
void MFWorldPrint(const MFWorld* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  fprintf(stream, "(");
  MFModelStatusPrint(MFWorldStatus(that), stream);
  fprintf(stream, ") values[");
  for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
    fprintf(stream, "%f", MFWorldGetValue(that, iActor));
    if (iActor < MF_NBMAXACTOR - 1)
      fprintf(stream, ",");
  }
  fprintf(stream, "]");
  fprintf(stream, " forecast[");
  for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
    fprintf(stream, "%f", MFWorldGetForecastValue(that, iActor));
    if (iActor < MF_NBMAXACTOR - 1)
      fprintf(stream, ",");
  }
  fprintf(stream, "]");
}

// Print the MFTransition 'that' on the stream 'stream'
void MFTransitionPrint(const MFTransition* const that,
  FILE* const stream) {
  #if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  fprintf(stream, "transition from (");
  MFModelStatusPrint(
    MFWorldStatus(MFTransitionFromWorld(that)), stream);
  fprintf(stream, ") to (");
  if (MFTransitionToWorld(that) != NULL)
    MFModelStatusPrint(
      MFWorldStatus(MFTransitionToWorld(that)), stream);
  else
    fprintf(stream, "<null>");
  fprintf(stream, ") through (");
  MFModelTransitionPrint((MFModelTransition*)that, stream);
  fprintf(stream, ") values[");
  for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
    fprintf(stream, "%f", that->_values[iActor]);
    if (iActor < MF_NBMAXACTOR - 1)
      fprintf(stream, ",");
  }
  fprintf(stream, "]");
}
```

```
// Print the MFWorld 'that' and its MFTransition on the stream 'stream'
void MFWorldTransPrintln(const MFWorld* const that,
  FILE* const stream) {
  #if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  MFWorldPrint(that, stream);
  printf("\n");
  for (int iTrans = 0; iTrans < MFWorldGetNbTrans(that); ++iTrans) {
    fprintf(stream, "  ");
    MFTransitionPrint(MFWorldTransition(that, iTrans), stream);
    fprintf(stream, "\n");
  }
}

// Set the current world of the MiniFrame 'that' to match the
// MFModelStatus 'status'
// If the world is in computed worlds reuse it, else create a new one
// If we create a new one here and there are already has many computed
// worlds as the memory limit, free the current one to make room for
// the new one
void MFSetCurWorld(MiniFrame* const that,
  const MFModelStatus* const status) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (status == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'status' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a flag to memorize if we have found the world
  bool flagFound = false;
  // Declare a flag to manage the deletion of element in the set of
  // computed worlds
  bool moved = false;
  // Loop on computed worlds
  GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
  do {
    MFWorld* world = GSetIterGet(&iter);
    moved = false;
    // If this is the current world
    if (MFModelStatusIsSame(MFWorldStatus(world), status)) {
      // Ensure that the status is exactly the same by copying the
      // MFModelStatus struct, in case MFModelStatusIsSame refers only
      // to a subset of properties of the MFModelStatus
      memcpy(world, status, sizeof(MFModelStatus));
      // Update the curWorld in MiniFrame
      that->_curWorld = world;
```

```
      flagFound = true;
    // Else, if it's a disposable world
    } else if (MFModelStatusIsDisposable(MFWorldStatus(world),
      MFWorldStatus(MFCurWorld(that)), MFGetNbComputedWorld(that))) {
      // Remove this world from its sources
      while (GSetNbElem(MFWorldSources(world)) > 0) {
        MFTransition* transSource = MFWorldPopSource(world);
        MFTransitionSetToWorld(transSource, NULL);
      }
      // Remove this world from the sources of its next worlds
      for (int iTrans = MFWorldGetNbTrans(world); iTrans--;) {
        const MFTransition* trans = MFWorldTransition(world, iTrans);
        MFWorld* toWorld = (MFWorld*)MFTransitionToWorld(trans);
        if (toWorld != NULL)
          MFWorldRemoveSource(toWorld, trans);
      }
      // Remove this world from the computed worlds
      moved = GSetIterRemoveElem(&iter);
      // Free memory
      MFWorldFree(&world);
    }
  } while (moved || GSetIterStep(&iter));
  // If we haven't found the searched status
  if (!flagFound) {
    // Create a new MFWorld with the current status
    MFWorld* world = MFWorldCreate(status);
    // Get the sente for the previous world
    int sente = MFModelStatusGetSente(MFWorldStatus(MFCurWorld(that)));
    // Add it to the computer worlds
    MFAddWorld(that, world, sente);
    // Update the current world
    that->_curWorld = world;
  }
}

// Remove the MFTransition 'source' from the sources of the
// MFWorld 'that'
void MFWorldRemoveSource(MFWorld* const that,
  const MFTransition* const source) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (source == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'source' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Loop on transitions
  if (GSetNbElem(MFWorldSources(that)) > 0) {
    bool moved = false;
    GSetIterForward iter =
      GSetIterForwardCreateStatic(MFWorldSources(that));
    do {
      moved = false;
      MFTransition* trans = GSetIterGet(&iter);
      if (trans == source) {
        moved = GSetIterRemoveElem(&iter);
      }
```

```
    } while (moved || GSetIterStep(&iter));
  }
}


// Pop a MFTransition from the sources of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFTransition* MFWorldPopSource(MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return GSetPop(&(that->_sources));
}
```

2.2 miniframe-inline.c

```
// ============ MINIFRAME_INLINE.C ================

// ================ Functions implementation ====================

// Get the time limit for expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetMaxTimeExpansion(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_maxTimeExpansion;
}

// Get the time unused during last expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeUnusedExpansion(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_timeUnusedExpansion;
}

// Get the time used to search world to expand during next expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
```

```
inline
#endif
float MFGetTimeSearchWorld(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_timeSearchWorld;
}

// Get the nb of world expanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldExpanded(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_nbWorldExpanded;
}

// Get the time used at end of expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeEndExpansion(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_timeEndExpansion;
}

// Get the clock considered has start during expansion
#if BUILDMODE != 0
inline
#endif
clock_t MFGetStartExpandClock(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_startExpandClock;
}

// Set the clock considered has start during expansion to 'c'
#if BUILDMODE != 0
inline
```

```
#endif
void MFSetStartExpandClock(MiniFrame* const that, clock_t c) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_startExpandClock = c;
}

// Set the time limit for expansion of the MiniFrame 'that' to
// 'timeLimit', in millisecond
// The time is measured with the function clock(), see "man clock"
// for details
#if BUILDMODE != 0
inline
#endif
void MFSetMaxTimeExpansion(MiniFrame* const that, const float timeLimit) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_maxTimeExpansion = timeLimit;
}

// Get the current MFWorld of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFCurWorld(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_curWorld;
}

// Get the GSet of computed MFWorlds of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorlds(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return &(that->_worlds);
}

// Add the MFWorld 'world' to the computed MFWorlds of the
// MiniFrame 'that', ordered by the world's value from the pov of
```

```
// actor 'iActor'
#if BUILDMODE != 0
inline
#endif
void MFAddWorld(MiniFrame* const that, \
  const MFWorld* const world, const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (world == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'world' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < -1 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (-1<=%d<%d)",
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  GSetAddSort(&(that->_worlds), world,
    MFWorldGetPOVValue(world, iActor));
}

// Return the MFModelStatus of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFModelStatus* MFWorldStatus(const MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return (const MFModelStatus*)that;
}

// Get the number of transition for the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
int MFWorldGetNbTrans(const MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_nbTransition;
}

// Get the percentage of resued world of the MiniFrame 'that' during
// the last MFEpxand()
#if BUILDMODE != 0
inline
```

```
#endif
float MFGetPercWordReused(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_percWorldReused;
}

// Return true if the MFTransition 'that' is expandable, i.e. its
// 'toWorld' is null, else return false
#if BUILDMODE != 0
inline
#endif
bool MFTransitionIsExpandable(const MFTransition* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return (that->_toWorld == NULL ? true : false);
}

// Get the 'iTrans' MFTransition of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFTransition* MFWorldTransition(const MFWorld* const that,
  const int iTrans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iTrans <0 || iTrans >= that->_nbTransition) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
      iTrans, that->_nbTransition);
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_transitions + iTrans;
}

// Compute the MFModelStatus resulting from the 'iTrans' MFTransition
// of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFModelStatus MFWorldComputeTransition(const MFWorld* const that,
  const int iTrans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
```

```
  }
  if (iTrans <0 || iTrans >= that->_nbTransition) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
      iTrans, that->_nbTransition);
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Return the resulting MFModelStatus
  return MFModelStatusStep((const MFModelStatus* const)that,
    (const MFModelTransition* const)MFWorldTransition(that, iTrans));
}

// Return true if the expansion algorithm looks in previously
// computed worlds for same world to reuse, else false
#if BUILDMODE != 0
inline
#endif
bool MFIsWorldReusable(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_reuseWorld;
}

// Set the falg controlling if the expansion algorithm looks in
// previously computed worlds for same world to reuse to 'reuse'
#if BUILDMODE != 0
inline
#endif
void MFSetWorldReusable(MiniFrame* const that, const bool reuse) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_reuseWorld = reuse;
}

// Get the MFWorld which the MFTransition 'that' is leading to
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionToWorld(const MFTransition* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_toWorld;
}

// Set the MFWorld to which the MFTransition 'that' is leading to
// 'world'
```

```
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetToWorld(MFTransition* const that,
  MFWorld* const world) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_toWorld = world;
}

// Get the MFWorld which the MFTransition 'that' is coming from
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionFromWorld(const MFTransition* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_fromWorld;
}

// Set the value of the MFTransition 'that' for the actor 'iActor' to
// 'val'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetValue(MFTransition* const that, const int iActor,
  const float val) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)",
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_values[iActor] = val;
}

// Return the number of computed worlds in the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbComputedWorld(const MiniFrame* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
```

```
    PBErrCatch(MiniFrameErr);
  }
#endif
  return GSetNbElem(&(that->_worlds));
}

// Return the egocentric value of the MFTransition 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFTransitionGetValue(const MFTransition* const that,
  const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_values[iActor];
}

// Return the egocentric value of the MFWorld 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFWorldGetValue(const MFWorld* const that, const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_values[iActor];
}

// Get the set of MFTransition reaching the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorldSources(const MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
```

```
#endif
  return &(that->_sources);
}

// Return the array of values of the MFWorld 'that' for each actor
#if BUILDMODE != 0
inline
#endif
const float* MFWorldValues(const MFWorld* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_values;
}
```

3 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: main

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Path to the model implementation
MF_MODEL_PATH=$(ROOT_DIR)/MiniFrame/Examples/BasicExample

# Rules to make the executable
repo=miniframe
$($(repo)_EXENAME): \
createLinkToModelHeader \
miniframe-model.o \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(MF_MODEL_PATH)/miniframe-model

$($(repo)_EXENAME).o: \
$(MF_MODEL_PATH)/miniframe-model.h \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$

createLinkToModelHeader:
ln -s -f $(MF_MODEL_PATH)/miniframe-model.h $($(repo)_DIR)/miniframe-model.h; ln -s -f $(MF_MODEL_PATH)/miniframe-inl

miniframe-model.o: \
$(MF_MODEL_PATH)/miniframe-model.h \
$(MF_MODEL_PATH)/miniframe-model.c \
```

```
Makefile
$(COMPILER) $(BUILD_ARG) -c $(MF_MODEL_PATH)/miniframe-model.c
```

4 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"
#include "miniframe.h"

#define RANDOMSEED 0

void UnitTestMFTransitionCreateFree() {

  MFWorld world;
  MFModelTransition trans = {._move = 1};
  MFTransition act = MFTransitionCreateStatic(&world, &trans);
  if (act._fromWorld != &world ||
    act._toWorld != NULL ||
    memcmp(&(act._transition), &(trans),
      sizeof(MFModelTransition)) != 0) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionCreateStatic failed");
    PBErrCatch(MiniFrameErr);
  }
  for (int iActor = MF_NBMAXACTOR; iActor--;)
    if (ISEQUALF(act._values[iActor], 0.0) == false) {
      MiniFrameErr->_type = PBErrTypeUnitTestFailed;
      sprintf(MiniFrameErr->_msg, "MFTransitionCreateStatic failed");
      PBErrCatch(MiniFrameErr);
    }
  MFTransitionFreeStatic(&act);

  printf("UnitTestMFTransitionCreateFree OK\n");
}

void UnitTestMFTransitionIsExpandable() {

  MFWorld world;
  MFModelTransition trans = {._move = 1};
  MFTransition act = MFTransitionCreateStatic(&world, &trans);
  if (!MFTransitionIsExpandable(&act)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionIsExpandable failed");
    PBErrCatch(MiniFrameErr);
  }
  act._toWorld = &world;
  if (MFTransitionIsExpandable(&act)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionIsExpandable failed");
    PBErrCatch(MiniFrameErr);
  }
```

```
  MFTransitionFreeStatic(&act);

  printf("UnitTestMFTransitionIsExpandable OK\n");
}

void UnitTestMFTransitionGetSet() {
  MFWorld worldFrom;
  MFWorld worldTo;
  MFModelTransition trans = {._move = 1};
  MFTransition act = MFTransitionCreateStatic(&worldFrom, &trans);
  act._toWorld = &worldTo;
  if (MFTransitionToWorld(&act) != &worldTo) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionToWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFTransitionFromWorld(&act) != &worldFrom) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionFromWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  MFTransitionSetValue(&act, 0, 1.0);
  if (ISEQUALF(act._values[0], 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionSetValue failed");
    PBErrCatch(MiniFrameErr);
  }
  if (ISEQUALF(MFTransitionGetValue(&act, 0), 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionGetValue failed");
    PBErrCatch(MiniFrameErr);
  }
  if (ISEQUALF(MFTransitionGetPOVValue(&act, 0), 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionGetPOVValue failed");
    PBErrCatch(MiniFrameErr);
  }
  MFWorld worldB;
  MFTransitionSetToWorld(&act, &worldB);
  if (MFTransitionToWorld(&act) != &worldB) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFTransitionSetToWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  MFTransitionFreeStatic(&act);

  printf("UnitTestMFTransitionGetSet OK\n");
}

void UnitTestMFTransition() {
  UnitTestMFTransitionCreateFree();
  UnitTestMFTransitionIsExpandable();
  UnitTestMFTransitionGetSet();
  printf("UnitTestMFTransition OK\n");
}

void UnitTestMFWorldCreateFree() {

  MFModelStatus modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
  MFWorld* world = MFWorldCreate(&modelWorld);
  if (world == NULL ||
    GSetNbElem(&(world->_sources)) != 0 ||
```

```
    world->_nbTransition != 3) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldCreate failed");
    PBErrCatch(MiniFrameErr);
  }
  float val[MF_NBMAXACTOR] = {0.0};
  val[0] = -1.0;
  for (int iActor = MF_NBMAXACTOR; iActor--;)
    if (ISEQUALF(world->_values[iActor], val[iActor]) == false) {
      MiniFrameErr->_type = PBErrTypeUnitTestFailed;
      sprintf(MiniFrameErr->_msg, "MFWorldCreate failed");
      PBErrCatch(MiniFrameErr);
    }
  MFWorldFree(&world);
  if (world != NULL) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldFree failed");
    PBErrCatch(MiniFrameErr);
  }

  printf("UnitTestMFWorldCreateFree OK\n");
}

void UnitTestMFWorldGetSet() {
  MFModelStatus modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
  MFWorld* world = MFWorldCreate(&modelWorld);
  if (MFWorldStatus(world) != &(world->_status)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldStatus failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFWorldGetNbTrans(world) != 3) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldGetNbTrans failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFWorldSources(world) != &(world->_sources)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldSources failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFWorldValues(world) != world->_values) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldValues failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFWorldTransition(world, 0) != world->_transitions ||
    MFWorldTransition(world, 1) != world->_transitions + 1 ||
    MFWorldTransition(world, 2) != world->_transitions + 2) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldTransition failed");
    PBErrCatch(MiniFrameErr);
  }
  world->_values[0] = 1.0;
  if (ISEQUALF(MFWorldGetValue(world, 0), 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldGetValue failed");
    PBErrCatch(MiniFrameErr);
  }
  MFWorldFree(&world);
  printf("UnitTestMFWorldGetSet OK\n");
}
```

```
void UnitTestMFWorldComputeTransition() {
  MFModelStatus modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
  MFWorld* world = MFWorldCreate(&modelWorld);
  MFModelStatus statusA = {._step = 1, ._pos = -1, ._tgt = 1};
  MFModelStatus statusB = {._step = 1, ._pos = 0, ._tgt = 1};
  MFModelStatus statusC = {._step = 1, ._pos = 1, ._tgt = 1};
  MFModelStatus status = MFWorldComputeTransition(world, 0);
  if (memcmp(&status, &statusA, sizeof(MFModelStatus)) != 0) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
    PBErrCatch(MiniFrameErr);
  }
  status = MFWorldComputeTransition(world, 1);
  if (memcmp(&status, &statusB, sizeof(MFModelStatus)) != 0) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
    PBErrCatch(MiniFrameErr);
  }
  status = MFWorldComputeTransition(world, 2);
  if (memcmp(&status, &statusC, sizeof(MFModelStatus)) != 0) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
    PBErrCatch(MiniFrameErr);
  }
  MFWorldFree(&world);
  printf("UnitTestMFWorldComputeTransition OK\n");
}

void UnitTestMFWorld() {
  UnitTestMFWorldCreateFree();
  UnitTestMFWorldGetSet();
  UnitTestMFWorldComputeTransition();
  printf("UnitTestMFWorld OK\n");
}

void UnitTestMiniFrameCreateFree() {

  MFModelStatus initStatus = {._step = 0, ._pos = 0, ._tgt = 1};
  MiniFrame* mf = MiniFrameCreate(&initStatus);
  if (mf == NULL ||
    mf->_nbStep != 0 ||
    ISEQUALF(mf->_maxTimeExpansion, MF_DEFAULTTIMEEXPANSION) == false ||
    MFModelStatusIsSame(&initStatus, &(MFCurWorld(mf)->_status)) == false ||
    MFCurWorld(mf) != GSetGet(MFWorlds(mf), 0) ||
    GSetNbElem(MFWorlds(mf)) != 1 ||
    ISEQUALF(mf->_timeUnusedExpansion, 0.0) == false ||
    ISEQUALF(mf->_percWorldReused, 0.0) == false ||
    mf->_nbWorldExpanded != 0 ||
    mf->_timeEndExpansion <= 0.0 ||
    mf->_reuseWorld != false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MiniFrameCreate failed");
    PBErrCatch(MiniFrameErr);
  }
  MiniFrameFree(&mf);
  if (mf != NULL) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MiniFrameFree failed");
    PBErrCatch(MiniFrameErr);
  }
```

```
  printf("UnitTestMiniFrameCreateFree OK\n");
}

void UnitTestMiniFrameGetSet() {
  MFModelStatus initWorld = {._step = 0, ._pos = 0, ._tgt = 1};
  MiniFrame* mf = MiniFrameCreate(&initWorld);
  if (ISEQUALF(MFGetMaxTimeExpansion(mf),
    mf->_maxTimeExpansion) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetMaxTimeExpansion failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFGetNbComputedWorld(mf) != 1) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetNbComputedWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  float t = MF_DEFAULTTIMEEXPANSION + 1.0;
  MFSetMaxTimeExpansion(mf, t);
  if (ISEQUALF(MFGetMaxTimeExpansion(mf), t) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetMaxTimeExpansion failed");
    PBErrCatch(MiniFrameErr);
  }
  if (ISEQUALF(MFGetTimeEndExpansion(mf),
    mf->_timeEndExpansion) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetTimeEndExpansion failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFCurWorld(mf) != mf->_curWorld) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFCurWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFWorlds(mf) != &(mf->_worlds)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorlds failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFIsWorldReusable(mf) != mf->_reuseWorld) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFIsWorldReusable failed");
    PBErrCatch(MiniFrameErr);
  }
  bool reuse = !MFIsWorldReusable(mf);
  MFSetWorldReusable(mf, reuse);
  if (MFIsWorldReusable(mf) != reuse) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetWorldReusable failed");
    PBErrCatch(MiniFrameErr);
  }
  mf->_percWorldReused = 1.0;
  if (ISEQUALF(MFGetPercWordReused(mf), 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetPercWordReused failed");
    PBErrCatch(MiniFrameErr);
  }
  MFModelStatus modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
  MFWorld* world = MFWorldCreate(&modelWorld);
  MFAddWorld(mf, world, -1);
  if (GSetNbElem(MFWorlds(mf)) != 2 ||
```

```
    MFModelStatusIsSame(MFWorldStatus(world),
      (MFModelStatus*)GSetGet(MFWorlds(mf), 1)) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFAddWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  mf->_nbWorldExpanded = 1;
  if (MFGetNbWorldExpanded(mf) != mf->_nbWorldExpanded) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetNbWorldExpanded failed");
    PBErrCatch(MiniFrameErr);
  }
  mf->_timeSearchWorld = 2.0;
  if (ISEQUALF(MFGetTimeSearchWorld(mf),
    mf->_timeSearchWorld) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetTimeSearchWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  mf->_timeUnusedExpansion = 3.0;
  if (ISEQUALF(MFGetTimeUnusedExpansion(mf),
    mf->_timeUnusedExpansion) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetTimeUnusedExpansion failed");
    PBErrCatch(MiniFrameErr);
  }
  mf->_percWorldReused = 4.0;
  if (ISEQUALF(MFGetPercWordReused(mf),
    mf->_percWorldReused) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetPercWordReused failed");
    PBErrCatch(MiniFrameErr);
  }
  clock_t now = clock();
  MFSetStartExpandClock(mf, now);
  if (mf->_startExpandClock != now) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetStartExpandClock failed");
    PBErrCatch(MiniFrameErr);
  }
  if (MFGetStartExpandClock(mf) != now) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetStartExpandClock failed");
    PBErrCatch(MiniFrameErr);
  }
  MiniFrameFree(&mf);
  printf("UnitTestMiniFrameGetSet OK\n");
}

void UnitTestMiniFrameExpandSetCurWorld() {
  MFModelStatus initWorld = {._step = 0, ._pos = 0, ._tgt = 2};
  MiniFrame* mf = MiniFrameCreate(&initWorld);
  MFSetStartExpandClock(mf, clock());
  MFSetWorldReusable(mf, true);
  MFExpand(mf);
  printf("Time unused by MFExpand: %f\n", MFGetTimeUnusedExpansion(mf));
  printf("Time search world to expand: %f\n", MFGetTimeSearchWorld(mf));
  printf("Nb world expanded: %d\n", MFGetNbWorldExpanded(mf));
  printf("Perc world reused: %f\n", MFGetPercWordReused(mf));
  printf("Computed worlds:\n");
  GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(mf));
  do {
```

```
    MFWorld* world = GSetIterGet(&iter);
    MFWorldTransPrintln(world, stdout);
  } while (GSetIterStep(&iter));
  if (mf->_timeUnusedExpansion < 0.0 ||
    MFGetNbWorldExpanded(mf) != 16 ||
    ISEQUALF(MFGetPercWordReused(mf), 0.625) == false ||
    ISEQUALF(MFGetTimeSearchWorld(mf), 100.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFExpand failed");
    PBErrCatch(MiniFrameErr);
  }
  const MFModelTransition* bestTrans = MFGetBestTransition(mf, 0);
  printf("Best action: %d\n", bestTrans->_move);
  if (bestTrans->_move != 1) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetBestTransition failed");
    PBErrCatch(MiniFrameErr);
  }
  if (ISEQUALF(MFWorldGetPOVValue(MFCurWorld(mf), 0), -2.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldGetPOVValue failed");
    PBErrCatch(MiniFrameErr);
  }
  if (ISEQUALF(
    MFWorldGetForecastValue(MFCurWorld(mf), 0), -0.00001) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldGetForecastValue failed");
    PBErrCatch(MiniFrameErr);
  }
  MFModelStatus nextWorld = {._pos = -1, ._tgt = 2};
  MFSetCurWorld(mf, &nextWorld);
  if (MFCurWorld(mf) != GSetGet(MFWorlds(mf), 1) ||
    MFGetNbComputedWorld(mf) != 4) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetCurWorld failed");
    PBErrCatch(MiniFrameErr);
  }
  MiniFrameFree(&mf);
  printf("UnitTestMiniFrameExpandSetCurWorld OK\n");
}

void UnitTestMiniFrameFullExample() {
  // Initial world
  MFModelStatus curWorld = {._step = 0, ._pos = 0, ._tgt = 2};
  // Create the MiniFrame
  MiniFrame* mf = MiniFrameCreate(&curWorld);
  // Set reusable worlds
  MFSetWorldReusable(mf, true);
  // Loop until end of game
  int tgt[7] = {2,2,-1,-1,-1,-1,-1};
  while (!MFModelStatusIsEnd(&curWorld)) {
    // Set the start clock
    MFSetStartExpandClock(mf, clock());
    // Correct the current world in the MiniFrame
    MFSetCurWorld(mf, &curWorld);
    // Expand
    MFExpand(mf);
    // Get best transition
    const MFModelTransition* bestTrans = MFGetBestTransition(mf, 0);
    if (bestTrans != NULL) {
      // Step with best transition
      curWorld = MFModelStatusStep(&curWorld, bestTrans);
```

```
    }
    // Apply external forces to the world
    curWorld._tgt = tgt[curWorld._step];
    // Display the current world
    printf("mf(");
    MFModelStatusPrint(MFWorldStatus(MFCurWorld(mf)), stdout);
    printf(") real(");
    MFModelStatusPrint(&curWorld, stdout);
    printf(")\n");
  }
  MiniFrameFree(&mf);
  printf("UnitTestMiniFrameFullExample OK\n");
}

void UnitTestMiniFrame() {
  UnitTestMiniFrameCreateFree();
  UnitTestMiniFrameGetSet();
  UnitTestMiniFrameExpandSetCurWorld();
  UnitTestMiniFrameFullExample();
  printf("UnitTestMiniFrame OK\n");
}

void UnitTestAll() {
  UnitTestMFTransition();
  UnitTestMFWorld();
  UnitTestMiniFrame();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

5 Unit tests output

```
UnitTestMFTransitionCreateFree OK
UnitTestMFTransitionIsExpandable OK
UnitTestMFTransitionGetSet OK
UnitTestMFTransition OK
UnitTestMFWorldCreateFree OK
UnitTestMFWorldGetSet OK
UnitTestMFWorldComputeTransition OK
UnitTestMFWorld OK
UnitTestMiniFrameCreateFree OK
UnitTestMiniFrameGetSet OK
Time unused by MFExpand: 99.963997
Time search world to expand: 100.000000
Nb world expanded: 16
Perc world reused: 0.625000
Computed worlds:
(step:3 pos:-3 tgt:2) values[-5.000000] forecast[-0.000100]
  transition from (step:3 pos:-3 tgt:2) to (step:3 pos:-3 tgt:2) through (move:-1) values[-0.000150]
  transition from (step:3 pos:-3 tgt:2) to (step:3 pos:-3 tgt:2) through (move:0) values[-0.000150]
  transition from (step:3 pos:-3 tgt:2) to (step:2 pos:-2 tgt:2) through (move:1) values[-0.000100]
(step:2 pos:-2 tgt:2) values[-4.000000] forecast[-0.000070]
  transition from (step:2 pos:-2 tgt:2) to (step:3 pos:-3 tgt:2) through (move:-1) values[-0.000150]
```

```
  transition from (step:2 pos:-2 tgt:2) to (step:2 pos:-2 tgt:2) through (move:0) values[-0.000100]
  transition from (step:2 pos:-2 tgt:2) to (step:1 pos:-1 tgt:2) through (move:1) values[-0.000070]
(step:1 pos:-1 tgt:2) values[-3.000000] forecast[-0.000040]
  transition from (step:1 pos:-1 tgt:2) to (step:2 pos:-2 tgt:2) through (move:-1) values[-0.000100]
  transition from (step:1 pos:-1 tgt:2) to (step:1 pos:-1 tgt:2) through (move:0) values[-0.000070]
  transition from (step:1 pos:-1 tgt:2) to (step:0 pos:0 tgt:2) through (move:1) values[-0.000040]
(step:0 pos:0 tgt:2) values[-2.000000] forecast[-0.000010]
  transition from (step:0 pos:0 tgt:2) to (step:1 pos:-1 tgt:2) through (move:-1) values[-0.000070]
  transition from (step:0 pos:0 tgt:2) to (step:0 pos:0 tgt:2) through (move:0) values[-0.000040]
  transition from (step:0 pos:0 tgt:2) to (step:1 pos:1 tgt:2) through (move:1) values[-0.000010]
(step:1 pos:1 tgt:2) values[-1.000000] forecast[0.000000]
  transition from (step:1 pos:1 tgt:2) to (step:0 pos:0 tgt:2) through (move:-1) values[-0.000040]
  transition from (step:1 pos:1 tgt:2) to (step:1 pos:1 tgt:2) through (move:0) values[-0.000010]
  transition from (step:1 pos:1 tgt:2) to (step:2 pos:2 tgt:2) through (move:1) values[-0.000000]
(step:2 pos:2 tgt:2) values[-0.000000] forecast[0.000000]
Best action: 1
UnitTestMiniFrameExpandSetCurWorld OK
mf(step:0 pos:0 tgt:2) real(step:1 pos:1 tgt:2)
mf(step:1 pos:1 tgt:2) real(step:2 pos:2 tgt:-1)
mf(step:2 pos:2 tgt:-1) real(step:3 pos:1 tgt:-1)
mf(step:3 pos:1 tgt:-1) real(step:4 pos:0 tgt:-1)
mf(step:4 pos:0 tgt:-1) real(step:5 pos:-1 tgt:-1)
UnitTestMiniFrameFullExample OK
UnitTestMiniFrame OK
UnitTestAll OK
```

6 Examples

6.1 Basic example

6.1.1 miniframe-model.h

```
// ============ MINIFRAME_MODEL.H ================

// As an example the code below implements a world where one actor
// moves along a discrete axis by step of one unit to reach a fixed
// target position
// Status of the world is defined by the current actor position and
// the target position
// Available actions are -1, 0, +1 (next position = current position
// + action) if the actor hasn't reached the target, else no actions
// The position of the actor is bounded to -5, 5
// The value of the world is given by -abs(position-target)

// ================= Include ==================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "/home/bayashi/GitHub/PBErr/pberr.h"

// ================= Define ==================

// Max number of actors in the world
// must be at least one
```

```
#define MF_NBMAXACTOR 1
// Max number of transitions possible from any given status
// must be at least one
#define MF_NBMAXTRANSITION 3

// ================= Data structure ==================

// Structure describing the transition from one instance of
// MFModelStatus to another
typedef struct MFModelTransition {
  int _move;
} MFModelTransition;

// Structure describing the status of the world at one instant
typedef struct MFModelStatus {
  int _step;
  int _pos;
  int _tgt;
} MFModelStatus;

// ================ Functions declaration ====================

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
  MFModelStatus* const tho);

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that);

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that);

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
  const MFModelStatus* const tho);

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that);

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that,
  const int iActor);

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
  MFModelTransition* const transitions, int* const nbTrans);
```

```
// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and  memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
  float* const values);

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
  const MFModelTransition* const trans);

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
  FILE* const stream);

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
  FILE* const stream);

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when whould
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
  const MFModelStatus* const curStatus, const int nbStatus);

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that);

// Init the board
void MFModelStatusInit(MFModelStatus* const that);

#if BUILDMODE != 0
inline
#endif
void toto();

// ================ Inliner ====================

#if BUILDMODE != 0
#include "miniframe-inline-model.c"
#endif
```

6.1.2 miniframe-model.c

```
// ============ MINIFRAME_MODEL.C ================

// As an example the code below implements a world where one actor
// moves along a discrete axis by step of one unit to reach a fixed
// target position
// Status of the world is defined by the current actor position and
// the target position
// Available actions are -1, 0, +1 (next position = current position
// + action) if the actor hasn't reached the target, else no actions
// The position of the actor is bounded to -5, 5
```

```
// The value of the world is given by -abs(position-target)

// ================= Include =================

#include "miniframe-model.h"
#if BUILDMODE == 0
#include "miniframe-inline-model.c"
#endif

// ================ Functions implementation ====================

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
  MFModelStatus* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (tho == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'tho' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)memcpy(tho, that, sizeof(MFModelStatus));
}

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that) {
  (void)that;

}

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that) {
  (void)that;

}

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
  const MFModelStatus* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (tho == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'tho' is null");
    PBErrCatch(MiniFrameErr);
  }
```

```
#endif
  if (that->_pos == tho->_pos &&
    that->_tgt == tho->_tgt)
    return true;
  else
    return false;
}

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)that;
  return 0;
}

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that, const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)that;(void)iActor;

  return true;
}

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
  MFModelTransition* const transitions, int* const nbTrans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (transitions == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'transitions' is null");
    PBErrCatch(MiniFrameErr);
```

```
  }
  if (nbTrans == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'nbTrans' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  if (that->_pos == that->_tgt) {
    *nbTrans = 0;
  } else {
    *nbTrans = 3;
    transitions[0]._move = -1;
    transitions[1]._move = 0;
    transitions[2]._move = 1;
  }
}

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and  memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
  float* const values) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (values == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'values' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  values[0] = -1.0 * fabs(that->_tgt - that->_pos);
}

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
  const MFModelTransition* const trans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (trans == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'trans' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the resulting status
  MFModelStatus status;
  // Apply the transition
  status._step = that->_step + 1;
  status._tgt = that->_tgt;
  status._pos = that->_pos + trans->_move;
  int limit = 3;
  if (status._pos < -limit) status._pos = -limit;
```

```
  if (status._pos > limit) status._pos = limit;
  // Return the status
  return status;
}

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  fprintf(stream, "step:%d pos:%d tgt:%d", that->_step,
    that->_pos, that->_tgt);
}

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  fprintf(stream, "move:%d", that->_move);
}

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when whould
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
  const MFModelStatus* const curStatus, const int nbStatus) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (curStatus == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'curStatus' is null");
    PBErrCatch(MiniFrameErr);
```

```
  }
#endif
  if (nbStatus > 0) {
    if (abs(that->_pos - curStatus->_pos) > 2)
      return true;
    else
      return false;
  } else {
    return false;
  }
}

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  if (that->_step >= 6 || that->_pos == that->_tgt) {
    return true;
  } else {
    return false;
  }
}
```

6.1.3 miniframe-inline-model.c

```
// ============ MINIFRAME-INLINE-MODEL.C ================

// ================ Functions implementation ====================

#if BUILDMODE != 0
inline
#endif
void toto() {

}
```

6.2 Oware

6.2.1 miniframe-model.h

```
// ============ MINIFRAME_MODEL.H ================

// ================= Include ==================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "/home/bayashi/GitHub/PBErr/pberr.h"

// ================= Define ==================
```

```
// Current implementation doesn't allow more than 2 players
// due to undefined end condition
#define NBPLAYER 2
#define NBHOLEPLAYER 6
#define NBHOLE (NBHOLEPLAYER * NBPLAYER)
#define NBINITSTONEPERHOLE 4
#define NBSTONE (NBHOLE * NBINITSTONEPERHOLE)
#define NBMAXTURN 500

// Max number of actors in the world
// must be at least one
#define MF_NBMAXACTOR NBPLAYER
// Max number of transitions possible from any given status
// must be at least one
#define MF_NBMAXTRANSITION NBHOLEPLAYER

// ================= Data structure ===================

// Structure describing the transition from one instance of
// MFModelStatus to another
typedef struct MFModelTransition {
  // Index of the hole from where stones are moved by the current player
  int _iHole;
} MFModelTransition;

// Structure describing the status of the world at one instant
typedef struct MFModelStatus {
  int _nbTurn;
  int _nbStone[NBHOLE];
  int _score[NBPLAYER];
  // Flag for special end condition
  char _end;
  // Index of the player who has the sente
  int _curPlayer;
} MFModelStatus;

// ================ Functions declaration ====================

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
  MFModelStatus* const tho);

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that);

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that);

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
  const MFModelStatus* const tho);

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
```

```
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that);

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that,
  const int iActor);

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
  MFModelTransition* const transitions, int* const nbTrans);

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and  memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
  float* const values);

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
  const MFModelTransition* const trans);

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
  FILE* const stream);

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
  FILE* const stream);

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when whould
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
  const MFModelStatus* const curStatus, const int nbStatus);

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that);

// Init the board
void MFModelStatusInit(MFModelStatus* const that);

#if BUILDMODE != 0
inline
#endif
void toto();

// ================ Inliner ====================
```

```
#if BUILDMODE != 0
#include "miniframe-inline-model.c"
#endif
```

6.2.2 miniframe-model.c

```
// ============ MINIFRAME_MODEL.C ================

// ================= Include =================

#include "miniframe-model.h"
#if BUILDMODE == 0
#include "miniframe-inline-model.c"
#endif

#define rnd() (double)(rand())/(float)(RAND_MAX)

// ================ Functions implementation ====================

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
  MFModelStatus* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (tho == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'tho' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)memcpy(tho, that, sizeof(MFModelStatus));
}

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that) {
  (void)that;

}

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that) {
  (void)that;

}

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
  const MFModelStatus* const tho) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (tho == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'tho' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  bool ret = true;
  if (that->_curPlayer != tho->_curPlayer ||
    that->_end != tho->_end)
    ret = false;
  for (int iPlayer = NBPLAYER; iPlayer-- && ret;)
    if (that->_score[iPlayer] != tho->_score[iPlayer])
      ret = false;
  for (int iHole = NBHOLE; iHole-- && ret;)
    if (that->_nbStone[iHole] != tho->_nbStone[iHole])
      ret = false;
  return ret;
}

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  return that->_curPlayer;
}

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that, const int iActor) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
    MiniFrameErr->_type = PBErrTypeInvalidArg;
    sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
      iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)that;(void)iActor;
  // Incorrect if NBPLAYER > 2
  return true;
}

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
```

```
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
  MFModelTransition* const transitions, int* const nbTrans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (transitions == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'transitions' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (nbTrans == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'nbTrans' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  *nbTrans = 0;
  for (int iHole = that->_curPlayer * NBHOLEPLAYER;
    iHole < (that->_curPlayer + 1) * NBHOLEPLAYER;
    ++iHole) {
    if (that->_nbStone[iHole] > 0) {
      transitions[*nbTrans]._iHole = iHole;
      ++(*nbTrans);
    }
  }
}

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and  memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
  float* const values) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (values == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'values' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  for (int iPlayer = NBPLAYER; iPlayer--;) {
    values[iPlayer] = that->_score[iPlayer];
    values[iPlayer] += rnd() * 0.001;
  }
  if (MFModelStatusIsEnd(that)) {
    int iWinner = -1;
    for (int iPlayer = NBPLAYER; iPlayer--;) {
      if (iWinner == -1 || values[iWinner] < values[iPlayer]) {
        iWinner = iPlayer;
      }
    }
    values[iWinner] = 100.0;
```

```
  }
}

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
  const MFModelTransition* const trans) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (trans == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'trans' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  // Declare a variable to memorize the resulting status
  MFModelStatus status;
  // Apply the transition

  MFModelStatusCopy(that, &status);
  int nbStone = status._nbStone[trans->_iHole];
  // Remove stones from starting hole
  status._nbStone[trans->_iHole] = 0;
  // Distribute stones
  int jHole = trans->_iHole;
  while (nbStone > 0) {
    ++jHole;
    if (jHole == NBHOLE) jHole = 0;
    // Jump over starting hole
    if (jHole == trans->_iHole) ++jHole;
    if (jHole == NBHOLE) jHole = 0;
    ++(status._nbStone[jHole]);
    --nbStone;
  }
  // Check for captured stones
  char flagCaptured = 0;
  while ((jHole < status._curPlayer * NBHOLEPLAYER ||
    jHole >= (status._curPlayer + 1) * NBHOLEPLAYER) &&
    (status._nbStone[jHole] == 2 ||
    status._nbStone[jHole] == 3)) {
    status._score[status._curPlayer] += status._nbStone[jHole];
    status._nbStone[jHole] = 0;
    flagCaptured = 1;
    --jHole;
  }
  // Check for special end conditions
  // First, check that the opponent is not starving
  int nbStoneOpp = 0;
  for (int iHole = 0; iHole < NBHOLE; ++iHole) {
    if (iHole < status._curPlayer * NBHOLEPLAYER ||
      iHole >= (status._curPlayer + 1) * NBHOLEPLAYER)
      nbStoneOpp += status._nbStone[iHole];
  }
  // If the opponent is starving
  if (nbStoneOpp == 0) {
    if (flagCaptured == 1) {
      // If there has been captured stones, it means the current
      // player has starved the opponent. The current player looses.
```

```
      status._end = 1;
      status._score[status._curPlayer] = -100.0;
    } else {
      // If there was no captured stones, it means the opponent
      // starved itself. The current player catches all his own stones.
      status._end = 1;
      for (int iHole = 0; iHole < NBHOLE; ++iHole) {
        if (iHole >= status._curPlayer * NBHOLEPLAYER &&
          iHole < (status._curPlayer + 1) * NBHOLEPLAYER)
          status._score[status._curPlayer] +=
            status._nbStone[iHole];
      }
    }
  }
  // Step the current player
  ++(status._curPlayer);
  if (status._curPlayer == NBPLAYER)
    status._curPlayer = 0;
  // Increment the nb of turn
  ++(status._nbTurn);


  // Return the status
  return status;
}

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  fprintf(stream, "#%d: ", that->_nbTurn);
  for (int iHole = 0; iHole < NBHOLE; ++iHole)
    fprintf(stream, "%d ", that->_nbStone[iHole]);
  fprintf(stream, "\nscore: ");
  for (int iPlayer = 0; iPlayer < NBPLAYER; ++iPlayer)
    fprintf(stream, "%d:%d ", iPlayer, that->_score[iPlayer]);
}

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
  FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
```

```
  }
#endif
  fprintf(stream, "move:%d", that->_iHole);
}

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when whould
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
  const MFModelStatus* const curStatus, const int nbStatus) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
  if (curStatus == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'curStatus' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  (void)nbStatus;
  int nbRemainStoneCurStatus = 0;
  for (int iHole = NBHOLE; iHole--;)
    nbRemainStoneCurStatus += curStatus->_nbStone[iHole];
  int nbRemainStone = 0;
  for (int iHole = NBHOLE; iHole--;)
    nbRemainStone += that->_nbStone[iHole];
  if (nbRemainStone > nbRemainStoneCurStatus)
    return true;
  else
    return false;
}

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  if (that->_end == 1 ||
    that->_nbTurn == NBMAXTURN)
    return true;
  bool ret = false;
  for (int iPlayer = NBPLAYER; iPlayer--;) {
    // Incorrect if NBPLAYER > 2
    if (that->_score[iPlayer] * 2 > NBSTONE)
      ret = true;
  }
  // For the case NBPLAYER > 2
  /*if (ret == false) {
    int nbRemainStone = 0;
```

```
    for (int iHole = NBHOLE; iHole-- && ret == false;)
      nbRemainStone += that->_nbStone[iHole];
    if (nbRemainStone == 0)
      ret = true;
  }*/
  return ret;
}

// Init the board
void MFModelStatusInit(MFModelStatus* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
  }
#endif
  that->_end = 0;
  for (int iPlayer = NBPLAYER; iPlayer--;)
    that->_score[iPlayer] = 0;
  for (int iHole = NBHOLE; iHole--;)
    that->_nbStone[iHole] = NBINITSTONEPERHOLE;
  that->_curPlayer = 0;
  that->_nbTurn = 0;
}
```

6.2.3 miniframe-inline-model.c

```
// ============ MINIFRAME-INLINE-MODEL.C ================

// ================ Functions implementation ====================

#if BUILDMODE != 0
inline
#endif
void toto() {

}
```

6.2.4 main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"
#include "miniframe.h"

#define RANDOMSEED 0

void RunGame() {
  // Initial world
  MFModelStatus curWorld;
  MFModelStatusInit(&curWorld);
  // Display the current world
```

```
  MFModelStatusPrint(&curWorld, stdout);
  printf("\n");
  // Create the MiniFrame
  MiniFrame* mf = MiniFrameCreate(&curWorld);
  // Set the expansion time
  MFSetMaxTimeExpansion(mf, 100.0);
  // Set reusable worlds
  MFSetWorldReusable(mf, true);
  // Flag to end the game
  bool flagEnd = false;
  // Loop until end of game
  while (!MFModelStatusIsEnd(&curWorld) && !flagEnd) {
    printf("-----------\n");
    // Set the start clock
    MFSetStartExpandClock(mf, clock());
    // Correct the current world in the MiniFrame
    MFSetCurWorld(mf, &curWorld);
    // Expand
    MFExpand(mf);
    // Display info about exansion
    printf("exp: %d ", MFGetNbWorldExpanded(mf));
    printf("comp: %d ", MFGetNbComputedWorld(mf));
    printf("unused: %fms\n", MFGetTimeUnusedExpansion(mf));
    if (MFGetTimeUnusedExpansion(mf) < 0.0)
      flagEnd = true;
    // Get best transition
    const MFModelTransition* bestTrans =
      MFGetBestTransition(mf, MFModelStatusGetSente(&curWorld));
    if (bestTrans != NULL) {
      // Display the transition
      MFModelTransitionPrint(bestTrans, stdout);
      printf("\n");
      // Step with best transition
      curWorld = MFModelStatusStep(&curWorld, bestTrans);
    } else {
      flagEnd = true;
    }
    // Apply external forces to the world
    // curWorld. = ... ;
    // Display the current world
    MFModelStatusPrint(&curWorld, stdout);
    printf("\n");
    fflush(stdout);
  }
  MiniFrameFree(&mf);
}

int main() {
  RunGame();
  // Return success code
  return 0;
}
```

6.2.5 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1
```

```
all: main

# Makefile definitions
MAKEFILE_INC=../../../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Path to the model implementation
MF_MODEL_PATH=$(ROOT_DIR)/MiniFrame/Examples/Oware

# Rules to make the executable
main: \
createLinkToModelHeader \
main.o \
miniframe-model.o \
$(miniframe_EXE_DEP) \
$(miniframe_DEP)
$(COMPILER) ‘echo "$(miniframe_EXE_DEP) main.o" | tr ’ ’ ’\n’ | sort -u‘ miniframe-model.o $(LINK_ARG) $(miniframe_LI

main.o: \
main.c \
$(miniframe_INC_H_EXE) \
$(miniframe_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $(miniframe_BUILD_ARG) ‘echo "$(miniframe_INC_DIR)" | tr ’ ’ ’\n’ | sort -u‘ -c main.c

createLinkToModelHeader:
ln -s -f $(MF_MODEL_PATH)/miniframe-model.h $(miniframe_DIR)/miniframe-model.h; ln -s -f $(MF_MODEL_PATH)/miniframe-

miniframe-model.o: miniframe-model.h miniframe-model.c Makefile
$(COMPILER) $(BUILD_ARG) -c miniframe-model.c
```

6.2.6 Example

```
#0: 4 4 4 4 4 4 4 4 4 4 4 4
score: 0:0 1:0
-----------
exp: 580 comp: 2969 unused: 0.536000ms
move:2
#1: 4 4 0 5 5 5 5 4 4 4 4 4
score: 0:0 1:0
-----------
exp: 215 comp: 4062 unused: 0.541000ms
move:11
#2: 5 5 1 6 5 5 5 4 4 4 4 0
score: 0:0 1:0
-----------
exp: 162 comp: 4885 unused: 0.762000ms
move:3
#3: 5 5 1 0 6 6 6 5 5 5 4 0
score: 0:0 1:0
-----------
exp: 131 comp: 5568 unused: 0.838000ms
move:9
#4: 6 6 0 0 6 6 6 5 5 0 5 1
score: 0:0 1:2
-----------
exp: 116 comp: 6170 unused: 1.026000ms
move:5
#5: 6 6 0 0 6 0 7 6 6 1 6 0
score: 0:2 1:2
```

```
----------
exp: 534 comp: 2810 unused: 0.498000ms
move:9
#6: 6 6 0 0 6 0 7 6 6 0 7 0
score: 0:2 1:2
----------
exp: 216 comp: 3877 unused: 0.557000ms
move:1
#7: 6 0 1 1 7 1 8 7 6 0 7 0
score: 0:2 1:2
----------
exp: 161 comp: 4701 unused: 0.916000ms
move:10
#8: 7 1 2 2 8 0 8 7 6 0 0 1
score: 0:2 1:4
----------
exp: 141 comp: 5336 unused: 0.701000ms
move:1
#9: 7 0 3 2 8 0 8 7 6 0 0 1
score: 0:2 1:4
----------
exp: 448 comp: 3056 unused: 0.325000ms
move:8
#10: 8 1 4 2 8 0 8 7 0 1 1 2
score: 0:2 1:4
----------
exp: 234 comp: 4200 unused: 0.514000ms
move:1
#11: 8 0 5 2 8 0 8 7 0 1 1 2
score: 0:2 1:4
----------
exp: 180 comp: 5089 unused: 0.750000ms
move:10
#12: 8 0 5 2 8 0 8 7 0 1 0 3
score: 0:2 1:4
----------
exp: 144 comp: 5830 unused: 0.696000ms
move:0
#13: 0 1 6 3 9 1 9 8 1 1 0 3
score: 0:2 1:4
----------
exp: 134 comp: 6481 unused: 0.835000ms
move:6
#14: 1 2 7 4 9 1 0 9 2 2 1 4
score: 0:2 1:4
----------
exp: 121 comp: 7066 unused: 0.919000ms
move:2
#15: 1 2 0 5 10 2 1 10 0 0 1 4
score: 0:8 1:4
----------
exp: 123 comp: 7443 unused: 1.284000ms
move:7
#16: 2 3 1 6 11 0 1 0 1 1 2 5
score: 0:8 1:7
----------
exp: 619 comp: 2985 unused: 0.462000ms
move:3
#17: 2 3 1 0 12 1 2 1 0 0 2 5
score: 0:12 1:7
----------
exp: 254 comp: 4095 unused: 0.653000ms
```

```
move:10
#18: 0 3 1 0 12 1 2 1 0 0 0 6
score: 0:12 1:10
-----------
exp: 631 comp: 2820 unused: 0.337000ms
move:5
#19: 0 3 1 0 12 0 0 1 0 0 0 6
score: 0:15 1:10
-----------
exp: 254 comp: 3860 unused: 0.553000ms
move:7
#20: 0 3 1 0 12 0 0 0 1 0 0 6
score: 0:15 1:10
-----------
exp: 549 comp: 2981 unused: 0.492000ms
move:4
#21: 1 4 2 1 0 2 1 1 2 1 1 7
score: 0:15 1:10
-----------
exp: 288 comp: 4097 unused: 0.761000ms
move:8
#22: 1 4 2 1 0 2 1 1 0 2 2 7
score: 0:15 1:10
-----------
exp: 203 comp: 4957 unused: 0.663000ms
move:5
#23: 1 4 2 1 0 0 0 0 0 2 2 7
score: 0:19 1:10
-----------
exp: 177 comp: 5688 unused: 0.793000ms
move:10
#24: 0 4 2 1 0 0 0 0 0 2 0 8
score: 0:19 1:12
-----------
exp: 535 comp: 3137 unused: 0.411000ms
move:1
#25: 0 0 3 2 1 1 0 0 0 2 0 8
score: 0:19 1:12
-----------
exp: 286 comp: 4213 unused: 0.454000ms
move:9
#26: 0 0 3 2 1 1 0 0 0 0 1 9
score: 0:19 1:12
-----------
exp: 231 comp: 5065 unused: 0.754000ms
move:5
#27: 0 0 3 2 1 0 1 0 0 0 1 9
score: 0:19 1:12
-----------
exp: 201 comp: 5778 unused: 0.716000ms
move:11
#28: 1 1 4 3 2 1 2 1 1 0 1 0
score: 0:19 1:12
-----------
exp: 150 comp: 6395 unused: 1.125000ms
move:2
#29: 1 1 0 4 3 2 0 1 1 0 1 0
score: 0:22 1:12
-----------
exp: 148 comp: 6962 unused: 1.460000ms
move:7
#30: 1 1 0 4 3 2 0 0 2 0 1 0
```

```
score: 0:22 1:12
-----------
exp: 782 comp: 2977 unused: 0.257000ms
move:5
#31: 1 1 0 4 3 0 1 1 2 0 1 0
score: 0:22 1:12
-----------
exp: 376 comp: 4101 unused: 0.492000ms
move:7
#32: 1 1 0 4 3 0 1 0 3 0 1 0
score: 0:22 1:12
-----------
exp: 254 comp: 4947 unused: 0.656000ms
move:1
#33: 1 0 1 4 3 0 1 0 3 0 1 0
score: 0:22 1:12
-----------
exp: 227 comp: 5659 unused: 0.571000ms
move:6
#34: 1 0 1 4 3 0 0 1 3 0 1 0
score: 0:22 1:12
-----------
exp: 172 comp: 6268 unused: 0.976000ms
move:4
#35: 1 0 1 4 0 1 1 0 3 0 1 0
score: 0:24 1:12
-----------
exp: 169 comp: 6823 unused: 1.288000ms
move:6
#36: 1 0 1 4 0 1 0 1 3 0 1 0
score: 0:24 1:12
-----------
exp: 314 comp: 4234 unused: 0.660000ms
move:3
#37: 1 0 1 0 1 2 1 0 3 0 1 0
score: 0:26 1:12
```