WP6 - Fault-Tolerance Components

Deliverable D6.2_rep - Fault tolerant components V2

WP6 - Fault-tolerance components : Deliverable 6.2_rep - Fault-tolerant components V2
by A. Lanusse and P. Vanuxeem

Published April 2004

Table of Contents

Document Presentation

Project Coordinator

Organisation:	UPVLC
Responsible person:	Alfons Crespo
Address:	Camino Vera, 14, 46022 Valencia, Spain
Phone:	+34 963877576
Fax:	+34 963877576
Email:	alfons@disca.upv.es

Participant List

Role	Id.	Participant Name	Acronym	Country
CO	1	Universidad Politecnica de Valencia	UPVLC	E
CR	2	Scuola Superiore Santa Anna	SSSA	I
CR	3	Czech Technical University in Prague	CTU	CZ
CR	4	CEA/DRT/LIST/DTSI	CEA	FR
CR	5	Unicontrols	UC	CZ
CR	6	MNIS	MNIS	FR
CR	7	Visual Tools S.A.	VT	E

Document version

Release	Date	Reason of change
1_0	04/05/03	First release
2_0	15/04/04	Second release

Chapter 1. Introduction

In this document, FT- or ft- prefixe means fault-tolerant or fault-tolerance.

The fault-tolerance components included in this deliverable consists of two complementary components (**ftappmon** and **ftcontroller**) which provide a framework for implementing degraded mode management support. Located at the application level, they provide both global monitoring of application and local control of execution. The current version handles only Hard real-time RTLinux level. The two components must be used together.

The following schema proposes a FT architecture overview.

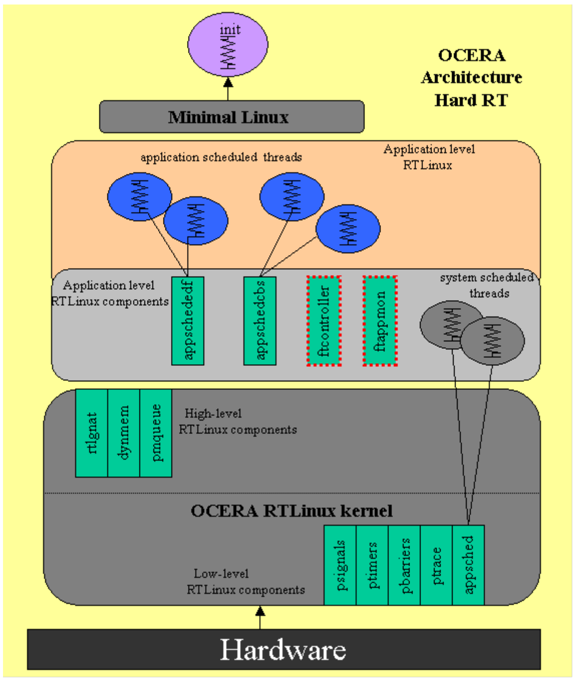

init
OCERA Architecture Hard RT
Minimal Linux
application scheduled threads
Application level RTLinux
Application level RTLinux components
appschededf
appschedcbs
ftcontroller
ftappmon
system scheduled threads
rtlgnat
dynmem
pmqueue
High-level RTLinux components
OCERA RTLinux kernel
Low-level RTLinux components
psignals
ptimers
pbarriers
ptrace
appsched
Hardware

Figure 1 . FT architecture overview

The **ftappmon** (application fault-tolerance monitor) component is in charge of applying application mode change on notification (from **ftcontroller**) of an abnormal situation related to a particular ft-task. A ft-task is a user task for which fault-tolerance is required. It involves two alternate threads, the first one implementing a nominal behavior and the second one implementing a degraded behavior. These two threads are created during the application init phase but only the nominal behavior is made active. The **ftappmon** defines the impact of the event on the current running tasks and decides of a new configuration (stops tasks, switches tasks modes, activates new tasks).

The **ftcontroller** component is a low-level RTLinux application component in charge of controlling execution of ft-tasks. On detection of an abnormal situation on a thread related to a ft-task (deadline miss or abort), the **ftcontroller** activates if possible the alternate thread (degraded behavior thread) and propagates the event to the **ftappmon**.

In the current implementation, we consider the implementation architecture at the OCERA Hard RT level only, and all the application tasks are hard real-time periodic tasks.

1.1 Basic interaction between components

The main interactions between components are illustrated in the following figure.

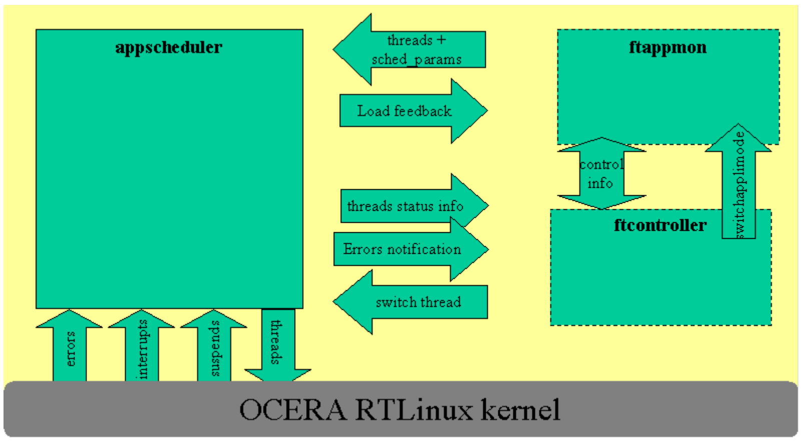


Figure 2 . Interactions with FT components

At init, information on application modes and tasks is dispatched between the different data structures and tables in the **ftappmon** and the **ftcontroller**.

This concerns the list of all tasks of the application and their associated behaviors along with the transition tables and error hooks.

When the application starts, threads are submitted with associated scheduling parameters to the appscheduler. When threads are elected by the scheduler for running, a notification is made towards the **ftcontroller** with the deadline parameter.

The **ftcontroller** keeps tracks of all threads started. If the threads end normally, the **ftcontroller** is informed. If an error occurs or a deadline is missed, a specific handler stored in a table in the **ftcontroller** applies immediate action. By default the action is to switch the active thread to his alternate suspended thread. Then, the **ftcontroller** notifies the **ftappmon** of the abnormal event and of task mode switch.

The **ftappmon** checks if the abnormal event triggers an application mode change condition and activates it if it is the case. This results in a new configuration of tasks behaviors and associated scheduling parameters to be submitted to the application scheduler.

The error management process can be illustrated by the next diagram.

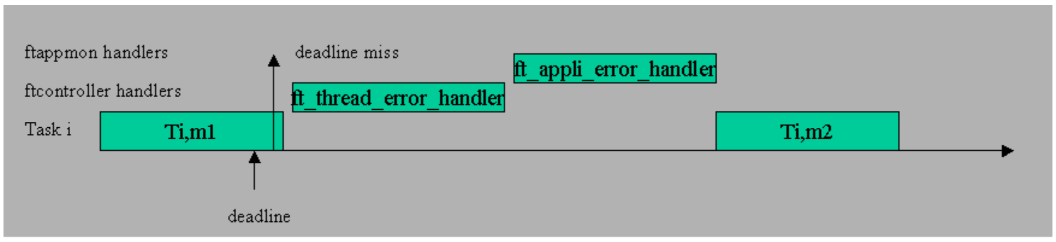


Figure 3 . Error management process

For more details on the general philosophy of fault-tolerance issues in OCERA please consult the Deliverable D6.1 (WP6 section).

1.2 Principles of utilisation

Using these two FT components requires an OCERA kernel implementation including at least the **ptrace** component, the **ftappmon** and the **ftcontroller.**

A pure fault tolerant application must use only Hard RTlinux tasks. Controlled threads are limited to threads associated to FT-tasks. An FT-task is an encapsulation that allows the transparent management of fault-tolerance.

The **FT API** (fault tolerance application programming interface) provides to the application developer specific FT functions which may be used in the application user code.

The **FT application model** specifies the ft-tasks behaviors, application modes and application modes transitions related to faulty events. Related FT API functions permits to provide to the FT components some information to achieve when needed degraded mode management at the task level. When a ft-task running with a normal behavior is reported faulty, the alternate degraded mode is then automatically activated.

The **FT API** provides also to the user specific functions to init, create and end the FT-tasks. The **ft_task_init** and **ft_task_create** functions permit the initialization of data related to ft-task and to create the related threads. The **ft_task_end** function permits to terminate a ft-task and its related threads.

The code of tasks is written as usual in start-routines. The only difference is that the user must define two behaviors for each ft-task. Two routines have to be defined for each ft-task, one corresponding to the code to be run as the normal behavior of the task and the other one to be used as a degraded behavior.

In the following example of application user code, a single FT-task is defined with two normal and degraded behaviors.

First, the FT API headers have to be included in the application user code. In complement, the header and source code corresponding to the FT application model have to be then included in global part and init_module part of the application user code.

```
// FT API include
#include "ft_api_common.h"
#include "ft_api_appmon_appli.h"
// FT application model include code (generated code by FT-Builder)#include
"ft_appli_model.h"

int init_module(void) {
// FT application model source code (generated code by FT-Builder)
#include "ft_appli_model.c"
...
}
```

Then, the **init_module** part may use the two functions **ft_task_init** and **ft_task_create** in order to initialise and create the ft-task.

```
int init_module(void) {
  ...
  // Init a ft-task
  // A ft-task has 2 threads with normal and degraded behavior
  ...
  task_id=ft_task_init(task_name,
                       normal_behavior_routine,
                       degraded_behavior_routine,
                       normal_routine_param,
                       degraded_routine_param,
                       normal_sched_param,
                       degraded_sched_param);

  // Create a ft-task with normal behavior
  // The normal thread is awaked each period
  // The degraded thread is waiting
  task_behavior=FT_TASK_NORMAL;
  ft_task_create(task_id,
                 task_behavior);
  ...
  return 0;
}
```

Inittialisation and creation of FT-task

The **ft_task_init** function instantiates internal tables of the FT components adding a new FT-task and data related. Some information have to be specified such as ft-task name, associated behavior routines, behavior routines parameters (arguments) and behavior scheduling parameters (priority, period, deadline, duration).

The **ft_task_create** function really creates the threads (encapsulating usual pthread_create function) and activates the normal one.

Then, the behaviors of the ft-task are defined, by the way of the routines ***normal_behavior_routine*** and ***degraded_behavior_routine*** . Note that at each behavior (normal or degraded) corresponds one different thread. For this small example, those two behaviors have almost the same code.

```
/***********************************************************************/
```

```
/*                                                                      */
/* FT Application task : normal behavior routine                        */
/*                                                                      */
/************************************************************************/
void normal_behavior_routine(void *arg) {
  int no_cycle=0;
  int j=0;
  int nloops=0;
  int task_id=0;

  task_id = (int) arg;
  rtl_printf("\nApplication : ft-task %d, thread %d started, normal behavior",
task_id, pthread_self());
   rtl_printf("\nApplication : ft-task %d, thread %d switching to wait, normal
behavior\n", task_id, pthread_self());

  // Infinite loop
  while(1) {
    // Wait make periodic or next period of the normal behavior thread
    pthread_wait_np();
    if (no_cycle == 0) rtl_printf("\nApplication : ft-task %d, thread %d
switching to running, normal_behavior\n", task_id, pthread_self());
    no_cycle++;
    rtl_printf("\nApplication : ft-task %d, thread %d, no_cycle %d,
normal_behavior\n", task_id, pthread_self(), no_cycle);
    // Timing loop
    nloops=5000;
    for (j=0; j<nloops; j++);
  } // end of 'while'

  return;
}
```

Normal and degraded behavior routines of a FT-task

```
/************************************************************************/
/*                                                                      */
/* FT Application task : degraded behavior routine                      */
/*                                                                      */
/************************************************************************/
void degraded_behavior_routine(void *arg) {
  int no_cycle=0;
  int j=0;
  int nloops=0;
  int task_id=0;

  task_id = (int) arg;
  rtl_printf("\nApplication : ft-task %d, thread %d started, degraded
behavior", task_id, pthread_self());
  rtl_printf("\nApplication : ft-task %d, thread %d switching to wait, degraded
behavior\n", task_id, pthread_self());

  // Infinite loop
  while(1) {
    // Wait make periodic or next period of the degraded behavior thread
    pthread_wait_np();
    if (no_cycle == 0) rtl_printf("\nApplication : ft-task %d, thread %d
switching to running, degraded behavior\n", task_id, pthread_self());
    no_cycle++;
    rtl_printf("\nApplication : ft-task %d, thread %d, no_cycle %d, degraded
behavior\n", task_id, pthread_self(), no_cycle);
    // Timing loop
    nloops=5000;
    for (j=0; j<nloops; j++);
  } // end of 'while'

  return;
}
```

Finally, the **cleanup_module** function uses the **ft_task_end** function to cleanup the threads related to a FT-task .

```
void cleanup_module(void) {
  // Delete a ft-task
  ft_task_end(task_id);
}
```

Cleanup of FT-task

For the definition of several FT-tasks, the user has just to make a loop on the ft_task_init, ft_task_create, and ft_task_end functions and to define two normal and degraded routines by ft-task with different names.

In complement, the application mode transition/switch, that is a more global mode change at application level than only a ft-task behavior change, requires that several configurations of ft-tasks be defined in a declarative way. A design/build tool named **FT-Builder** has been developped in order to support the user in this design stage. The ft-builder generates the **FT application model** (header and code).

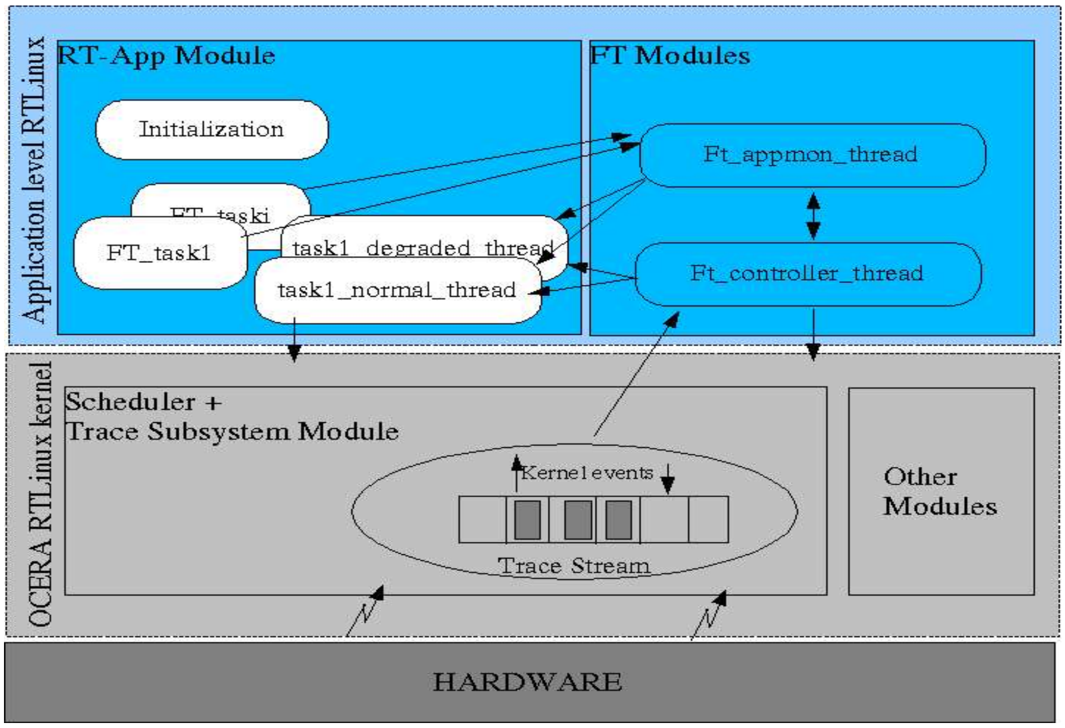
1.3 Current implementation principles

The main issue of the implementation is the detection of events related to threads and the possibility of reacting in a short and safe way to abnormal situations.

This implies a close cooperation with the scheduling component in order to :

- maintain a consistent and accurate image of the application (threads states);
- be informed of abnormal situations (abort or deadline miss as soon as they occur);
- be able to provide a replacement behavior in a deterministic way and in a deterministic time.

In the current implementation of ft-components, we have made the choice to work in a way that limits to the minimum the impact on other OCERA components (mainly scheduling components). So we have used the tracing facility offered by the OCERA **ptrace** component to survey the kernel events and react adequately to information gathered through this information source.



FT control architecture using the ptrace component

The **ptrace** component permits to track user and / or system events. The events are written by the scheduler into streams that can be read later on by specialized tasks. The **ptrace** component may also provide analysis and graphic support.

The **ftcontroller** is a periodic rtlinux task, it activates the kernel trace and reads system events related to threads (in the kernel stream created by the **ptrace** component on activation of tracing facility). It maintains a database of threads status and activates degraded behavior when abnormal situation is detected.

The advantage of this choice of implementation is that we can test ft-components independently of other components in a first place and validate the main architecture of the fault-tolerance framework.

The status of the implementation is still in a stable phase even if some tests have to be performed (verification). The example implemented tests degraded mode activation on detection of an abnormal event.

Detection of abort is simulated through detection of kill event. The abort event is not provided by the scheduler at the moment. This requires a deeper cooperation with the scheduler components development teams.

Detection of deadline_miss is not supported yet even if some development has started. The event is not provided by the scheduler used. In the future, if new scheduler components provide the event we can use it, if not, we can use the **ptimer** component to detect deadline_misses at the **ftcontroller** level.

Future extended development may concern :

1. Improved detection of abnormal events
2. Synchronization issues (atomicity and deterministic reaction time)
3. Continuation of development of application mode change facility (global).

The first point may be solved with close UPVLC cooperation and does not compromise the use of the **ptrace** component as support for event detection.

The second point has to be improved and may require a slight change of implementation strategy.

Actually, the main drawback of this first implementation is the fact that atomicity is lost between event occurrence at kernel level and handling in the **ftcontroller** since the detection mechanism relies currently on a polling strategy.

The only guaranteed property is that if an abnormal event occurs on a controlled thread, it can be detected and a reaction will take place within a delay bounded by the **ftcontroller** period (if the edfscheduler is used).

A more reactive implementation may be envisaged. First, an awaking pattern for ftcontroller is considered as a more appropriate mechanism than polling, where the ftcontroller will be waiting for an event to occur and react as soon as it can be read in the stream. A second possibility would be that when an abnormal situation is detected by the scheduler, the **ftcontroller** would be set to a high priority in order to handle the event and get back to its normal priority afterwards. A third one would be to let the scheduler use the **ftcontroller** API to handle the situation itself. The problem with that last solution would be that it would imply the writing of specialized FT versions of schedulers which would increase maintenance costs.

The third point has been seriously treated in the current implementation, in particular with the FT-Builder development. It could be of course enriched, in particular concerning resources (semaphore, mutex, fifo) management translation on different behaviors (threads) while application modes change.

1.4 Rapid overview of FT API

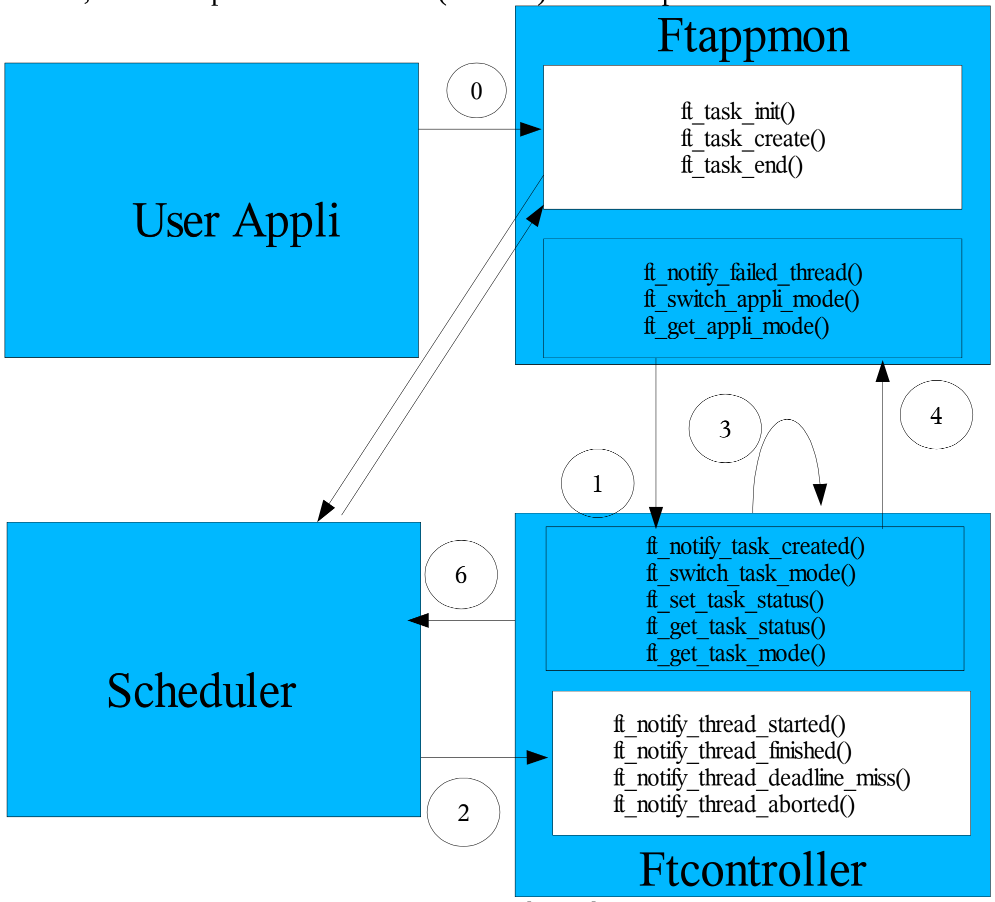
The Fault-Tolerance components described in this document have to be used jointly since they interfere strongly. It is the reason why though each one has its own API described in a distinct section, it may be useful to get a general overview of them.

Each component provides both external and internal API.

The **ftappmon** offers the application developer (external) API , named **FT API,** that is restricted to very few functions **ft_task_init()**, **ft_task_create()**, **ft_task_end()**. In complement, for FT application model, some additional API functions have been added : **ft_init_appli, ft_set_appli_mode, ft_set_appli_control**. The FT-Builder generating these function calls, these last functions may be considered as transparent to the user.

The **ftcontroller** has an (external) API that can be used by the scheduler mainly to notify events to the **ftcontroller**.

In addition, each component has also an (internal) API that permits interactions between them.



Interactions through FT APIs

In this schema, the application calls **ftappmon** external FT API functions at init (0), this induces notification to the **ftcontroller** (1), then threads are created. Each time the scheduler produces an event related to a ft_thread, a notification is issued (2) towards the **ftcontroller**. The information is used to update the database and/or to react to abnormal situation (3). In case of abnormal situation, a notification is issued from the **ftcontroller** towards the **ftappmon**(4). Then an application mode switch can be activated (5).

The next sections describe in details the two FT components.

Chapter 2. ftappmon component

2.1.Summary

- Name : ftappmon
- Description :
- Author (name and email) :
  A. Lanusse (agnes.lanusse@cea.fr)
  P. Vanuxeem (patrick.vanuxeem@cea.fr)
- Reviewer :
- Layer : application RTLinux Level.
- Version : V0.2
- Status : test
- Dependencies : ocera, requires ftcontroller component
- Release date : M2

2.2 Description

The **ftappmon** (or ftappmonitor) component is the FT Application Monitor. This component consists of one controlling thread, an application control database and an application status database.

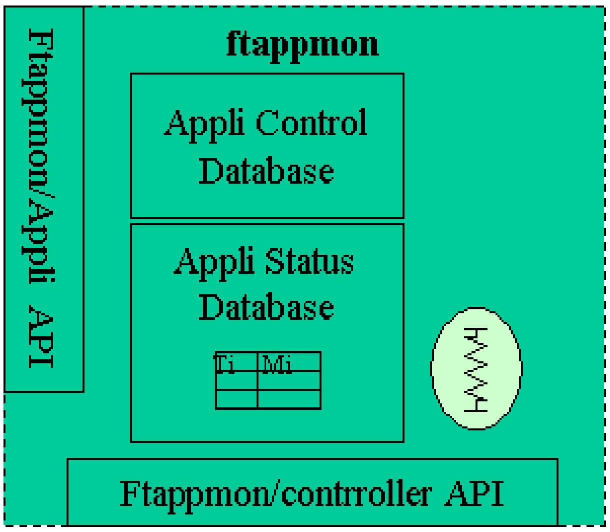


*Figure 4 . **ftappmon** : internal view*

The application control database contains the transition conditions for application mode change and the related actions.

Application mode change refers to dynamic reconfiguration of tasks in order to activate a new functioning of the application. This new functioning in our case is consecutive to partial failure of the application. The reconfiguration consists in redefining the set of eligible tasks in the new context and their associated behavior.

In the current implementation, predefined configurations are stored in the Appli Control Database and transition rules determine conditions of activation of a new mode.

The application status database contains information on the current configuration of the application, i.e. for each ft-task its status and active mode.

The ftappmon controlling thread receives notification of errors and ft-task behavior changes, analyses the situation and if necessary activates an application mode change.

Depending on the error and the current application mode, the **ftappmon** detects which application mode transition must be fired. Then it compares the current configuration and the new targeted one.

For each ft-task in the current configuration, the configuration change specification is :

If the ft-task is present in the new configuration with the same behavior, nothing is done for the ft-task and its current running thread.

If the ft-task is present in the new configuration with a different behavior, the current thread of the ft-task is aborted, and the alternate behavior is made active for the next period.

For each ft-task not in the current configuration, new ft-tasks are created with appropriate behavior.

Synchronization is ensured in order to respect the periodic functioning of the application. Tasks interrupted during the period in which the error occurs will have misbehaving during this period that will be tolerated by tasks with which they communicate.

2.3 API / Compatibility

This component uses POSIX compatible RTLinux API for threads manipulation. A few additional functions have been defined. Scheduling parameters will follow the structure required by the OCERA scheduling components (CBS and / or EDF) if used.

The API can be divided into two subsets. An external API used by application developers (FT appmon/appli API) and an internal API to be used for communications between **ftappmon** and **ftcontroller** (FT appmon/controller API).

FT appmon/appli API
ft_task_init
ft_task_create
ft_task_end
FT appmon/controller API
ft_notify_failed_thread

FT appmon/appli API

This part deals with the API functions between the ftappmon component and application code called by application.

ft_task_init // init a ft-task

Parameters :

in

ft_task_name

ft_normal_routine

ft_degraded_routine

ft_normal_param

ft_degraded_param

ft_normal_sched_param

ft_degraded_sched_param

out

ft_task_id

Description :

The **ft_task_init** function is used by application developer to init ft-task. When invoked, the **ftappmon** instantiates its ft-task database and setup related data structures. In particular, information relative to behaviorial routines will be stored in order to be further used to handle related threads in ft_task_create function. The function returns the ft-task identifier to be used by the other functions ft_task_create and ft_task_end.

Prototype :

```
extern int ft_task_init
 (
  char *ft_task_name,
  void *(ft_normal_routine)(void*),
  void *(ft_degraded_routine)(void*),
  int ft_normal_param,
  int ft_degraded_param,
  FT_sched_param ft_normal_sched_param,
  FT_sched_param ft_degraded_sched_param
 );
```

ft_task_create // create a ft-task

Parameters :

in

```
          ft_task_id
          ft_task_behavior <normal | degraded>
```

Description :

The **ft_task_create** function is used by application developer to create a ft-task. It really creates the two threads related to the ft-task, encapsulating the pthread_create primitive. The thread corresponding to the selected behavior is made active while the other is suspended. The internal ft-task state database is updated.

Prototype :

```
extern int ft_task_create
 (
  int ft_task_id,
  FT_task_behavior ft_task_behavior
 );
```

ft_task_end // end a ft-task

Parameters :

in

```
          ft_task_id
```

Description :

The **ft_task_end** function is used by application developer to end an ft-task and its associated threads.

Prototype :

```
extern int ft_task_end
 (
  int ft_task_id
 );
```

FT appmon/controller API

This part deals with the API functions between the ftappmon and ftcontroller components called by ftcontroller.

ft_notify_failed_thread // notify a failed thread

Parameters :

in

```
          ft_failed_thread_kid
          ft_failed_task_id
          ft_failure_parameters
          ft_thread_failure_cause  < error | deadline_miss >
          ft_task_behavior_when_failed
          ft_task_behavior_current
```

Description :

The **ft_notify_failed_thread** function is used by the **ftcontroller** to inform the **ftappmon** that an abnormal situation occurs on the thread ft_failed_thread_kid related to the ft-task and that a new behavior has been activated. The **ftappmon** updates its database and checks if this failure triggers a need for mode change. If it is the case, it calls the internal ft_switch_appli_mode function.

Prototype :

```
static int ft_notify_failed_thread
 (
  pthread_t ft_failed_thread_id,
  int ft_failed_task_id,
  int ft_failure_parameters,
  FT_thread_failure_cause ft_thread_failure_cause,
  FT_task_behavior ft_task_behavior_when_failed,
  FT_task_behavior ft_task_behavior_current
  );
```

FT appmon internal functions

This part deals with the internal functions of the ftappmon component called by ftappmon.

ft_select_new_appli_mode // select a new application mode

Parameters :

in

```
          ft_failed_thread_kid
          ft_failed_task_id
          ft_failure_parameters
          ft_thread_failure_cause  < error | deadline_miss >
          ft_task_behavior_when_failed
          ft_task_behavior_current
```

Description :

The **ft_select_new_appli_mode** function is used internally by the **ftappmon** on notification of a failed thread. This function applies the rules specified by the FT application model in order to select a new application mode.

Prototype :

```
static int ft_select_new_appli_mode
 (
  pthread_t ft_failed_thread_kid,
  int ft_failed_task_id,
  int ft_failure_parameters,
  FT_thread_failure_cause ft_thread_failure_cause,
  FT_task_behavior ft_task_behavior_when_failed,
  FT_task_behavior ft_task_behavior_current,
  FT_Appli_Mode* ft_dest_appli_mode_ptr
 );
```

ft_switch_appli_mode // switch to an application mode

Parameters :

in

```
          ft_failed_thread_kid
          ft_failed_task_id
          ft_failure_parameters
          ft_thread_failure_cause  < error | deadline_miss >
          ft_task_behavior_when_failed
          ft_task_behavior_current
          ft_dest_appli_mode
```

Description :

The **ft_switch_appli_mode** function is used internally by the **ftappmon** on notification of a failed thread. This function makes effective the new application mode, calls the API function **ft_switch_task_behavior** of the **ftcontroller** to change ft-tasks behaviors and updates its database.

Prototype :

```
static int ft_switch_appli_mode
 (
  pthread_t ft_failed_thread_kid,
  int ft_failed_task_id,
  int ft_failure_parameters,
  FT_thread_failure_cause ft_thread_failure_cause,
  FT_task_behavior ft_task_behavior_when_failed,
  FT_task_behavior ft_task_behavior_current,
  FT_Appli_Mode ft_dest_appli_mode
  );
```

2.4 Implementation issues

- Modifications to the existing RTLinux or Linux code

This component is a new one, there is no modification to existing RTLinux component.

- Modifications of types, structures,tables and databases.

Following modifications of types, structures, tables and databases have allowed to improve the structure and behavior of the ftappmon component.

- Types, structures,tables and databases for ftappmon (in ft_appli_monitor.h)

- The table of the ft-tasks (ftappmon point of view) : **MTT :**

```
FT_Task_Elt ft_tasks_tab[FT_TASKS_MAX_NB+1]
```

with :

```
typedef struct {
  int  ft_task_id;
  char ft_task_name[FT_TASK_NAME_MAX_LN];
  FT_task_state  ft_task_state;
  FT_task_behavior ft_task_behavior;
  void (*ft_normal_routine)(void*);
  void (*ft_degraded_routine)(void*);
  int ft_normal_param;
  int ft_degraded_param;
  FT-sched_param ft_normal_sched_param;
  FT_sched_param ft_degraded_sched_param;
  pthread_t ft_normal_thread_kid;
  pthread_t ft_degraded_thread_kid;
} FT_Task_Elt;
```

- The FT Appli Control Database **ACDB :**

```
FT_Appli_Control_Database_Elt ft_app_ctrl_db
[FT_APP_MODES_MAX_NB+1];
with :
typedef struct {
  FT_Appli_Mode ft_app_mode;
  FT_Event_Task_Appli_Mode_Elt ft_event_task_app_modes_tab
[FT_TASKS_MAX_NB+1];
} FT_Appli_Control_Database_Elt;
with :
typedef struct {
  int ft_task_id;
  char ft_task_name[FT_TASK_NAME_LN];
  FT_Appli_Mode ft_kill_normal_dst_app_mode;
  FT_Appli_Mode ft_deadline_miss_normal_dst_app_mode;
  FT_Appli_Mode ft_kill_degraded_dst_app_mode;
  FT_Appli_Mode ft_deadline_miss_degraded_dst_app_mode;
} FT_Event_Task_Appli_Mode_Elt;
```

- The FT Appli Mode Table **AMT :**

```
FT_Appli_Mode_Elt ft_app_modes_tab[FT_APP_MODES_MAX_NB+1];
with :
typedef struct {
  FT_Appli_Mode ft_app_mode;
  FT_Task_Behavior_Elt ft_task_behaviors_tab
[FT_TASKS_MAX_NB+1];
} FT_Appli_Mode_Elt;
with :
typedef struct {
  int ft_task_id;
  char ft_task_name[FT_TASK_NAME_MAX_LN];
  FT_Task_Behavior ft_task_behavior;
} FT_Task_Behavior_Elt;
```

- The FT Appli State Database **ASDB**

```
FT_Appli_State_Database_Elt ft_app_stat_db[FT_TASKS_MAX]_NB+1;
with :
typedef struct {
  int ft_task_id;
  FT_task_state ft_task_state;
  FT_task_behavior ft_task_behavior;
  } FT_Appli_State_Database_Elt;
```

- API common types to application, ftappmon and ftcontroller (in ft_api_common.h) :

- FT-Task behavior :

```
typedef enum {
  FT_TASK_BEHAVIOR_NOT_DEFINED,
  FT_TASK_NOT_STARTED,
  FT_TASK_NORMAL,
  FT_TASK_DEGRADED,
  FT_TASK_TERMINATED
  } FT_task_behavior;
```

- FT scheduling parameters of one behavior of a ft-task :

```
typedef enum {
  int prio;
  hrtime_t periiod;
  hrtime_t deadline;
  hrtime_t duration;
} FT_sched_param;
```

- FT Task Behavior Element :

```
typedef struct {
  int ft_task_id;
  char ft_task_name[FT_TASK_NAME_MAX_LN];
  FT_Task_Behavior ft_task_behavior;
} FT_Task_Behavior_Elt;
```

- FT Appli Mode :

```
typedef struct {
  int ft_app_mode_id;
  char ft_app_mode_name[FT_MODE_NAME_MAX_LN];
} FT_Appli_Mode;
```

- FT Event Task Appli Mode :

```
typedef struct {
  int ft_task_id;
  char ft_task_name[FT_TASK_NAME_LN];
  FT_Appli_Mode ft_kill_normal_dst_app_mode;
  FT_Appli_Mode ft_deadline_miss_normal_dst_app_mode;
  FT_Appli_Mode ft_kill_degraded_dst_app_mode;
  FT_Appli_Mode ft_deadline_miss_degraded_dst_app_mode;
} FT_Event_Task_Appli_Mode_Elt;
```

- Common types to ftappmon and ftcontroller (in ft_common.h) :

- FT task state :

```
typedef enum {
  FT_TASK_STATE_NOT_DEFINED,
  FT_TASK_CREATED,
  FT_TASK_RUNNING,
  FT_TASK_ENDED
} FT_task_state;
```

- FT thread type :

```
typedef enum {
  FT_THREAD_TYPE_NOT_DEFINED,
  FT_THREAD_NORMAL,
  FT_THREAD_DEGRADED
} FT_thread_type;
```

- FT thread failure cause :

```
typedef enum {
  FT_THREAD_EVENT_NOT_DEFINED,
  FT_THREAD_KILL,
  FT_THREAD_DEADLINE_MISS
 } FT_thread_failure_cause;
```

2.5 Tests and validation

2.5.1 Validation criteria

Validation criteria concern mainly functional qualitative issues .

Application initialization

Verification that all the threads related to ft-tasks are created correctly and that the data structures are updated.

Application mode change is effective and correct after activation.

Verification that the new configuration of ft-tasks has been loaded and is active.

Verification that no ft-tasks from the previous configuration is left in an undesired state.

2.5.2 Test 1

Initialisation procedures.

This test has been achieved and shows correct data initialisation.

2.5.3 Test 2

Application mode commutation.

This test has been achieved and shows correct application mode change.

2.5.4 Results and comments

Tests have been performed by several examples with several application modes, ft-tasks and behaviors. Faulty event is only kill on a particular thread (pthread_kill) corresponding to a particular behavior. The tests runs with a user example module and the two components ftappmon and ftcontroller.

The facilities tested concern initialization procedures, behavior commutations and application mode changes.

The initialization of ft-tasks initializes correctly the data structures and creates the threads associated to two behaviors (normal and degraded) for each ft-task..

The application mode transition test shows correct application mode change with related ft-tasks behaviors change.

2.6 Examples

2.6.1 How to run the examples

The examples developed are common to the two components, please refer to the **ftcontroller** examples section.

2.6.2 Description

The examples developed are common to the two components, please refer to the **ftcontroller** examples section.

2.6.3 Results and comments

The examples developed are common to the two components, please refer to the **ftcontroller** examples section.

2.7 Installation instructions

See section 3.7

Chapter 3. ftcontroller component

3.1.Summary

- Name : ftcontroller
- Description :
- Author (name and email) :
  A. Lanusse (agnes.lanusse@cea.fr)
  P. Vanuxeem (patrick.vanuxeem@cea.fr)
- Reviewer :
- Layer : application RTLinux Level.
- Version : V0.2
- Status : test
- Dependencies : ocera, requires Ptrace component and ftappmon component
- Release date : M2

3.2 Description

The **ftcontroller** component is the FT Controller. This component consists of one controlling thread, a threads status database and a table containing handler routines for tasks mode switch on errors. Additional ft-tasks table from the ftcontroller point of view has been added.

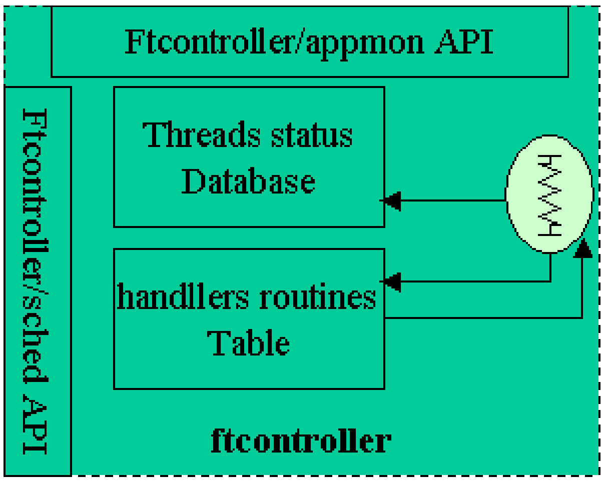


*Figure 5 . **ftcontroller** internal view*

The threads status data base collects the current status of the threads from the tracing facility (ptrace component). The ftcontroller reads periodically the kernel stream where kernel events are written by the scheduler.

The handlers routine table contains two entries for each possible thread of the application. These two entries correspond to the two types of errors (abort and deadline miss).

The routines have all the same structure and are instantiated at application init.

Each routine has four actions :

- Ends the current thread (faulty if not yet aborted)
- Sets the scheduling parameters for the replacement behavior
- Activates the replacement behavior
- Propagates event to ftappmon

The need for different routines resides in the fact that in the future, richer error handling strategies will be made available and may be different for different tasks.

3.3 API / Compatibility

This component uses posix compatible RTLinux API for threads manipulation and OCERA components API (ptrace, psignals, ptimers, pbarriers, appsched). A few additional functions have been defined.

The API can be divided into two subsets. An *external* API to be used for communications with the scheduler (FT controller/sched API) and an *internal* API to be used for communications between **ftappmon** and **ftcontroller** (FT controller/appmon API).

FT controller/sched API
ft_notify_thread_started
ft_notify_thread_finished
ft_notify_deadline_miss
ft_notify_thread_aborted
FT controller/appmon API
ft_notify_task_created
ft_switch_task_behavior
ft_set_task_status
ft_get_task_status
ft_get_task_behavior

FT controller/sched API

This part deals with the design API functions between ftcontroller and scheduler components called by scheduler.

ft_notify_thread_started

Parameters :

in :

ft_thread_kid

execution_start_date

deadline

Description:

The **ft_notify_thread_started** function could be used (not presently) by the scheduler to notify that a thread has started its execution (running and elected). If it is a periodic thread, it must be notified at each new period.

Prototype :

```
extern int ft_notify_thread_started
 (
  pthread_t ft_thread_kid,
  int execution_start_date,
  int deadline
 );
```

ft_notify_thread_finished

Parameters :

in :

ft_thread_kid

execution_end_date

deadline

Description :

The **ft_notify_thread_finished** function could be used (not presently) by the scheduler to notify the normal end of execution of a thread. For a periodic thread it notifies the end of a cycle.

Prototype :

```
extern int ft_notify_thread_finished
 (
  pthread_t ft_thread_kid,
  int execution_end_date,
  int deadline
 );
```

ft_notify_deadline_miss

Parameters :

in :

ft_thread_kid

Description :

The **ft_notify_deadline_miss** function could be used (not presently) by the scheduler to notify an error. This should rather be a signal. If not, the scheduler must set a high priority to **ftcontroller** after this call.

Prototype :

```
extern int ft_notify_deadline_miss
 (
  pthread_t ft_thread_kid
 );
```

ft_notify_thread_aborted // notify that a thread has aborted

Parameters :

in :

ft_thread_kid

ft_thread_failure_cause

Description :

The **ft_notify_thread_aborted** function could be used (not presently) by the scheduler to notify an error. This should rather be a signal. If not, the scheduler must set a high priority to **ftcontroller** after this call.

The **ft_notify_thread_aborted** function is used (presently) by the **ftcontroller** when pthread kill event is detected. This function calls the internal function ft_int_switch_task_behavior of the ftcontroller.

Prototype :

```
extern int ft_notify_thread_aborted
 (
  pthread_t ft_aborted_thread_kid,
  FT_thread_failure_cause ft_thread_failure_cause
  );
```

FT controller/appmon API

This part deals with the API functions between ftcontroller and ftappmon components called by ftappmon.

ft_notify_task_created // Notify that a ft-task has been created

Parameters :

in :

ft_task_id

ft_task_name

ft_task_state

ft_task_behavior

ft_normal_behavior_routine

ft_degraded_behavior_routine

ft_normal_sched_param

ft_degraded_sched_param

ft_normal_thread_kid

ft_degraded_thread_kid

Description :

The **ft_notify_task_created** function is used by the **ftappmon** to notify the creation of a ft-task and transmit related information to the **ftcontroller**. This information is used to init the data structures in the ftcontroller (threads status database, table of the ft-tasks).

Prototype :

```
int ft_notify_task_created
 (
  int  ft_task_id,
  char *ft_task_name,
  FT_task_state ft_task_state,
  FT_task_behavior ft_task_behavior,
  void (*ft_normal_behavior_routine)(void*),
  void (*ft_degraded_behavior_routine)(void*),
  pthread_t ft_normal_thread_kid,
  pthread_t ft_degraded_thread_kid
 );
```

ft_switch_task_behavior   // Switch the behavior of a ft-task

Parameters :

in :

ft_task_id

ft_task_src_behavior

ft_task_dst_behavior

Description :

The **ft_switch_task_behavior** function is used by the **ftappmon** to activate ft-task behavior switch. The ft-task behavior possible values are normal or degraded. It determines the current behavior of the ft-task. This function calls the internal function ft_int_switch_task_behavior of the ftcontroller.

Prototype :

```
extern int ft_switch_task_behavior
 (
   int ft_task_id,
   FT_task_behavior ft_task_src_behavior,
   FT_task_behavior ft_task_dst_behavior
 );
```

FT controller internal functions

This part deals with the internal functions of the ftcontroller component called by ftcontroller.

ft_int_switch_task_behavior   // Switch the behavior of a ft-task

Parameters :

in :

ft_normal_thread_id

ft_degraded_thread_id

ft_src_task_behavior

ft_dst_task_behavior

Description :

The **ft_int_switch_task_behavior** function is used internally by the **ftcontroller** to activate ft-task behavior switch. The ft-task behavior possible values are normal or degraded. It determines the current behavior of the ft-task.

Prototype :

```
int ft_int_switch_task_behavior
 (
   int ft_normal_thread_id,
   int ft_degraded_thread_id,
   FT_task_behavior ft_src_task_behavior,
   FT_task_behavior ft_dst_task_behavior
 );
```

ft_set_task_status

Parameters :

in :

```
          task_id
          task_status
```

Description :

The **ft_set_task_status** function could be used (not presently) internally by the **ftcontroller** to set the task status.

The task status tells if a task is created, active or ended.

Prototype :

```
int ft_set_task_status
 (
  int task_id,
  int task_status
 );
```

ft_get_task_status

Parameters :

in :

```
          task_id
```

out :

```
          task_status
```

Description :

The **ft_get_task_status** function could be used (not presently) internally by the **ftcontroller** to get the task status.

Prototype :

```
int ft_get_task_status
 (
  int task_id,
  int task_status
 );
```

ft_get_task_behavior

```
Parameters :

in :

          task_id

out :

          task_behavior


Description :
```

The **ft_get_task_behavior** function could be used (not presently) internally by the **ftcontroller** to get the task behavior.

The task behavior indicates the current active behavior of the task (normal or degraded).

```
Prototype :

int ft_get_task_behavior

 (

  int task_id,

  int task_mode

 );
```

3.4 Implementation issues

- Modifications to the existing RTLinux or Linux code

The ftcontroller component is a new one, there is no modification to existing RTLinux component.

- Modifications of types, structures, tables and databases.

Following modifications of types, structures, tables and databases have allowed to improve the structure and behavior of the ftcontroller component.

- Types, structure, tables and databases for ftcontroller (in ft_controller.h) :

- The FT thread event type

```
          typedef enum {

            FT_THREAD_CREATED,

            FT_THREAD_STARTED,

            FT_THREAD_STARTED_NEW_CYCLE,

            FT_THREAD_ENDED_CYCLE,

            FT_THREAD_ENDED_NORMALLY,

            FT_THREAD_ABORT,

          } FT_threads_event_types_t;
```

- The FT Threads Status Database : **TSDB**

```
          FT_Threads_Status_Database_Elt ft_thr_stat_db
```

```
[FT_THREADS_MAX_NB+1];
with :
typedef struct {
  pthread_t ft_thread_kid;
  FT_thread_type ft_thread_type;
  FT_sched_param ft_thread_sched_param;
  FT_thread_failure_cause ft_thread_failure_cause;
  int ft_task_id;
} FT_Threads_Status_Database_Elt;
```

- The table of the ft-tasks (ftcontroller point of view) : **CTT**

```
FT_task_elt_c ft_tasks_tab_c[FT_TASKS_MAX_NB+1]
```

with :

```
typedef struct {
  int  ft_task_id;
  char ft_task_name[FT_TASK_NAME_MAX_LN];
  FT_task_state ft_task_state;
  FT_task_behavior ft_task_behavior;
  void (*normal_behavior)(void*);
  void (*degraded_behavior)(void*);
  int ft_normal_thread_id;
  int ft_degraded_thread_id;
} FT_task_elt_c;
```

- The FT thread event

```
typedef struct {
  int       ft_thread_event_id;
  int       ft_thread_event_type;
  char      ft_thread_event_data;
  struct    timespec posix_timestamp;
  pthread_t ft_thread_id;
} FT_thread_event;
```

- API common types to application, ftappmon and ftcontroller (in ft_api_common.h) :

- See ftappmon component description.

- Common types to ftappmon and ftcontroller (in ft_common.h) :

- See ftappmon component description.

3.5 Tests and validation

3.5.1 Validation criteria

Validation criteria mainly concern functional qualitative criteria.

Verification that in absence of abnormal situation the application runs normally.

Verification that a faulty ft-task commutes correctly from a normal behavior to a degraded behavior (the normal thread is ended and the new one is started).

Verification that the propagation of an abnormal event to the **ftappmon** is achieved correctly.

Verification that faulty events (abort and deadline miss) are detected.

Further, we will verify synchronization issues.

Verification that the replacement behavior activation is achieved at the right time (next activation period of the previous running thread)

Verification of atomiticity of the commutation

and, if possible, performance issues will be targeted

Verification that commutation times satisfy minimum period requirement from the application.

3.5.2 Test 1

Management of ft-task commutation.

In this test, the behavior commutation of a ft-task on detection of an abnormal event is achieved. The abnormal event is simulated by a kill (issued by pthread_cancel).

On detection of the event, the concerned ft-task is identified, then the normal behavior is replaced by the degraded behavior. The thread running the degraded behavior, which was in a waiting state, is awakened and made running.

3.5.3 Test 2

Detection of abnormal event

The principle of abnormal event detection has been tested. It is based on the analysis of the kernel stream trace (ptrace component). For the test, the event looked for is a thread kill event.

With the version of scheduler used, no deadline miss or abort events are traced by the kernel. This should be provided.

3.5.4 Results and comments

The testing process is still on going.

Up to now, we have tested the basic communication mechanisms between the components involved in Fault-Tolerance management and the functioning of basic commutation mechanisms.

This has permitted to set up a global FT framework for degraded mode management. The principles of initialization, event detection, commutation and termination have been settled with success.

Anyway, the FT framework has to be improved by future extended development which may concern : improved detection of abnormal events, synchronization issues (atomicity and deterministic reaction time), continuation of development of global application mode change facility.

3.6 Example

The example provided is intended to test the different points cited above and related both to application initialization, event detection, behavior commutation, application switch and application termination.

Since there is no global example directory for Fault-Tolerance, all installation and testing are done in the *examples* subdirectory located within the **ftcontroller** component.

The example requires a running RTLinux version including the ptrace component. The components ptrace, ftappmon, ftcontroller, ftbuilder have to be selected previously at the OCERA installing phase (configuration).

3.6.1 Description

A simple fault-tolerant example named **ftappli** is proposed.

The **ftbuider** is used to specify and generate the application model. The application modes are FT_MODE_INIT, FT_MODE_DEGRADED, FT_MODE_STOP. The application mode FT_MODE_INIT is composed of two ft-tasks FT_TASK_1, FT_TASK_2 with respectively FT_TASK_NOT_STARTED, FT_TASK_NORMAL behaviors. The application mode FT_MODE_DEGRADED is composed of two ft-tasks FT_TASK_1, FT_TASK_2 with respectively FT_TASK_NORMAL, FT_TASK_DEGRADED behaviors. The application mode FT_MODE_STOP is composed of the two ft-tasks FT_TASK_1, FT_TASK_2 with FT_TASK_TERMINATED behaviors.

The example creates 2 periodic ft-tasks FT_TASK_1 and FT_TASK_2 with period equal to 1 s. Each ft-task has two behaviors : a normal behavior and a degraded behavior. The ft-task FT_TASK_1 has initially a not-started behavior (the ft-task is created but normal and degraded threads are suspended) whilst the ft-task FT_TASK_2 has a normal behavior. The 2 ft-tasks has the same normal and degraded behavior routines. The normal and degraded behaviors routines only performs loops.

On the 10th cycle, the normal behavior routine of the second ft-task FT_TASK_2 kills itself (by pthread_cancel(pthread_self()). Then the kill event is detected by **ftcontroller** and notified to **ftappmon.** Following the specified application model, the normal behavior of the first ft-task FT_TASK_1 and the degraded behavior of the ft-task FT_TASK_2 are activated. After a certain number of cycles in the new behaviors, the application is terminated.

This example is running during about 200 sec.

3.6.2 How to run the example

Description :

The example named **ftappli** is localed in the *ftappli* directory within the *examples* directory.

The Ocera installing phase (compilation) builds one module *ftappmonctrl.o* for the two FT components ftappmon and ftcontroller and also builds the example module *ftappli.o*.

The *Makefile* file located within the *ftappli* directory may build the example module *ftappli.o* (only if cleaned) and may instal (loading) and execute all the modules : FT components module *ftappmonctrl.o* and example module *ftappli.o* .

Implementation :

The *examples* directory (*ocera/components/ft/ftcontroller/examples)* is located within the **ftcontroller** component.

- The *examples* directory is composed of the following directories:

  - *ftappli*

and the following files:

  - *README*
  - *INSTALL*
  - *Makefile*

- The *ftappli* directory contains the following directories:

  - *include*
  - *src*

and the following files:

  - *README*
  - *INSTALL*
  - *Makefile*
  - *dmesg* or *messages*

Compilation :

Be careful : the compilation of the example has to be made only if *ftappli.o* module has been cleaned. If *ftappli.o* module already exists within *ftappli/src* directory, do-not make the compilation.

In order to compile the *ftappli.o* module (if cleaned), please follow next steps:

- Be a <*lambda*> user (not a root user)

```
$
```

- Go to the *ftappli* directory:

```
$ cd ocera/components/ft/ftcontroller/examples/ftappli
```

- Clean the *ftappli* directory:

```
$ make clean
```

- Compile the *ftappli* module:

```
$ make
```

<u>Installation/Execution :</u>

It is recommended to use the *Makefile* file in *ftappli* directory in order to install and execute all the modules.

Note that the installation order between modules is important due to the dependencies between the FT components (API).

The imperative order is:

- rtlinux start
- insmod ftappmonctrl.o
- insmod ftappli.o

In order to run the example, please follow next steps:

- Be a root user (necessary)

```
$ su
Password:
#
```

- Go to the ftappli directory:

```
# cd ocera/components/ft/ftcontroller/examples/ftappli
```

- Run the exemple:

The Makefile install and execute all the modules (ptrace, ft-components, example).

```
# make example
```

- Get the modules execution traces:

```
# dmesg > dmesg
```

```
or
```

```
# cp /var/log/messages .
# chown <owner> messages
# chgrp <group> messages
```

Be careful to see only the last execution traces (not the previous ones).

3.6.3 Results and comments

The execution trace (file *dmesg* or *messages*) shows the various events detected by the **ftcontroller** and the application switch and the commutation of behaviors can be seen.

The traces below show the threads creation at application initialisation and the application running. Whilst the ft-task 1 is not started, the ft-task 2 has normal behavior.

```
 mbuff: kernel shared memory driver v0.7.2 for Linux 2.4.18-ocera-1.0.0

 mbuff: (C) Tomasz Motylewski et al., GPL

 mbuff: registered as MISC device minor 254

 RTLinux Extensions Loaded (http://www.fsmlabs.com/)

 *****************

  FT_Controller

 ******************

 ******************

  FT_Appli_Monitor

 ******************

 **********

  FT_Appli

 **********

 Application : ap_i=2 ap_task_behavior=FT_TASK_NORMAL

 Application : ft-task 2, thread -951123968 started, normal or not
started behavior

 Application : ft-task 2, thread -951123968 switching to wait, normal or
not started behavior

 Application : ft-task 2, thread -951123968 switching to running,
normal_behavior

 Application : ap_i=1 ap_task_behavior=FT_TASK_NOT_STARTED

 Application : ft-task 1, thread -953942016 started, normal or not
started behavior

 Application : ft-task 1, thread -953942016 switching to wait, normal or
not started behavior
```

Whilst the ft-task 1 is not started, the ft-task 2 with normal thread runs during 10 cycles, then kills itself. The ftcontroller detects *pthread_kill* event, makes local change at thread level of the faulty behavior ft-task 2 and notifies to ftappmon the failed thread. The ftappmon selects new application mode and switches from *ft_mode_init* to *ft_mode_degraded* application mode. The ftappmon asks then to ftcontroller to really change at threads level the behaviors of the ft-tasks 1 and 2.

```
 Application : ft-task 2, thread -951123968 cancelling, normal_behavior

 FT_Controller : ft_task_id=2 PTHREAD_KILL

 FT_Controller : Cancel the normal thread !!!

                  ft task id               ---   2

                  ft normal thread kid     ---   -951123968

 Application : ft-task 2, thread -950566912 switching to running,
degraded_behavior

 FT_Controller : Make periodic the degraded thread !!!

                  ft task id               ---   2

                  ft degraded thread kid   ---   -950566912

 FT_Appli_Monitor : Function ft_notify_failed_thread

 FT_Appli_Monitor : before switch ft_current_appli_mode= {1 ,
FT_MODE_INIT}

 FT_Appli_Monitor : ft_new_appli_mode= {2 , FT_MODE_DEGRADED}

 Application : ft-task 1, thread -953942016 switching to running,
normal_behavior

 FT_Controller : Make periodic the normal thread !!!

                  ft task id               ---   1

                  ft normal thread kid     ---   -953942016

 FT_Controller : Cancel the normal thread !!!

                  ft task
```

The ft-task 2 with the degraded behvior is now running whilst the ft-task 1 has switched from not-started to normal behavior and is now running. The application is then stopped when 200 seconds are elapsed : the 2 ft-tasks are ended which imply that related threads are deleted. The modules of ptrace, example and ft-components are removed.

```
 Application : ft-task 2, thread -950566912, no_cycle 20,
degraded_behavior

 Application : ft-task 1, thread -953942016, no_cycle 20, normal_behavior

 Application : CLEANUP application threads !!!

 unloading mbuff

 mbuff device deregistered
```

3.7 Installation instructions

This the installation information and instructions for FT.

The two FT components **ftappmon** and **ftcontroller** provided make part of the OCERA tree under the *ft* branch.

This *ft* directory contains the following directories and files :

```
ft
|--- ftappmon
|    |--- README
|    |--- INSTALL
|    |--- Makefile
|    |--- doc
|    |--- include
|    |    |--- ft_api_appmon_appli
|    |    |--- ft_api_appmon_ctrl
|    |    |--- ft_api_common.h
|    |    |--- ft_appli_monitor.h
|    |--- src
|         |--- ft_appli_monitor.c
|--- ftcontroller
|    |--- README
|    |--- INSTALL
|    |--- Makefile
|    |--- include
|    |    |--- ft_api_ctrl_appmon.h
|    |    |--- ft_api_ctrl_sched.h
|    |    |--- ft_api_common.h
|    |    |--- ft_controller.h
|    |--- src
|    |    |--- Makefile
|    |    |--- ft_appmon_ctrl.c
|    |    |--- ft_controller.c
|    |--- examples
|         |--- INSTALL
|         |---  Makefile
|         |---  ftappli
|                |--- README
|                |--- INSTALL
|                |--- Makefile
|               |--- include
|               |    |--- ft_appli.h
|               |    |--- ft_appli_model.h
|               |--- src
|                    |--- Makefile
|                    |--- ft_appli.c
|                    |--- ft_appli_model.c
|--- ftbuilder
|--- ftredundancymgr
|--- ftreplicamgr
```

A general OCERA installation procedure is provided that install the selected components from a configuration selection. It makes use of the various makefile defined at each component level. This procedure has been tested and is globally operational for fault-tolerance.

For a separate testing of the FT components (ftappmon and ftcontroller) use the Makefile defined in *ftappli* directory (see example section above).

Chapter 4. ftbuilder component

4.1.Summary

- Name : ftbuilder
- Description : The FT-Builder is a design support tool that helps users' specify temporal constraints of their applications, modes definition and mode transitions conditions. Once this is done it generates code that is used to instanciate internal control databases of run-time FT components.
- Author (name and email) :
  A. Lanusse (agnes.lanusse@cea.fr)
  P. Vanuxeem (patrick.vanuxeem@cea.fr)
- Reviewer :
- Layer : Off-line Linux Component
- Version : V0.2
- Status : test
- Dependencies : Requires TCL/TK 8.3
- Release date : M2

4.2 Description

The design choices for FT facilities have been based on a declarative approach combined with transparent error handling mechanisms. This choice is driven by the fact that we consider fault-tolerance as non-functional requirements that must not interfere with application core coding for two main reasons: first to get a better control over consistency of fault-tolerance related coding and second, to facilitate maintainability since such requirements may be subject to change. Propagation of requirements change must be handled in a consistent manner which is much more complex if fault-tolerance programming is embedded in the user code.

According to these choices, non functional requirements related to fault-tolerance are collected through a design/build tool and used to instantiate the various run-time components in charge of the behavioural control of the application.

The **ftbuilder** is this design support tool that helps user's specify :

- temporal constraints of their applications,
- modes definition and
- mode transitions conditions.

Once this is done it generates code that is used to instanciate internal control databases of run-time FT components.

The approach retained for degraded mode management, relies on a specific programming model providing the concepts of ft_task and application_mode (along with the notions of ft_task_behaviour and application_mode_transition); and on two specific run-time components that implement degraded mode management through activation of ft_task_behaviour change and application_mode switching.

The role of these ft-components is to insure a transparent and safe management of such transitions at ft-task and application level. A particular attention has been paid to the overall application logical and temporal consistency and to a clean resource management so that aborting a task does not produce subsequent tasks blocking. The basic principles of degraded mode management according to this approach are the following: when an error is detected at task level, it triggers a task behaviour change to a degraded mode and propagates the notification of abnormal event at the application level where a decision is taken to apply or not an application mode change.

One of the major issue in the introduction of FT facilities was to preserve as far as possible user programming habits and thus to keep unchanged the way he writes tasks routines. We have thus introduced a limited number of primitives mainly used at init to declare what we call ft_tasks while the rest of code writing is kept unchanged. The only important thing concerning ft_tasks is that the user has to provide a routine for each possible behavior (actually two in the current implementation: one for the normal behavior and one for the degraded one).

The introduction of mode management at application level implies that additional information is provided to the system in order to handle abnormal situations in a proper way. This information is actually gathered into internal databases within the run-time ft_components.

In order to facilitate the initialization of these internal databases, information collected off-line is processed in order to produce specific files used at init to instantiate them. This way the user has not to provide additional code but only to include these files during the compilation of their application.

Development process

The proposed development process follows thus three steps :

- Global application design

  Done by user

  - identification and specification of the tasks and provides real_time constraints for each one
  - identification and specification of modes (valid configuration of tasks)
  - identification and specification of transitions between modes (transitions and conditions of transitions.

  Automatically generated

  Verification and code generation is achieved by the ftbuilder and provides

  - ft_appli_model.h
  - ft_appli_model.c

- Detailed task coding

  The user has to provide code for each task identified in the previous step .

  Actually he must provide two routines :

  - one for the normal_behavior

  - one for the degraded behavior

  Both routines are standard rtlinux routines that are run in rtlinux periodic threads

- Compilation

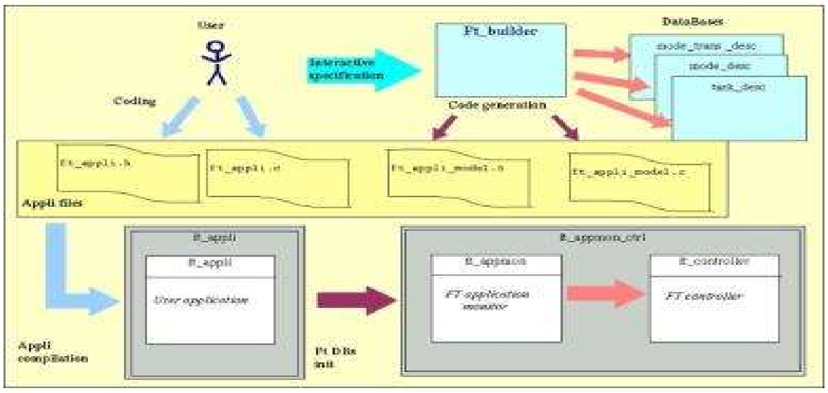
  Normal compilation using makefile, user' s application files are completed by inclusion of files generated by the Ftbuilder. This provides additionnal information that is used by the ft-components to control dynamically the application.

The first step is done with the Ftbuilder

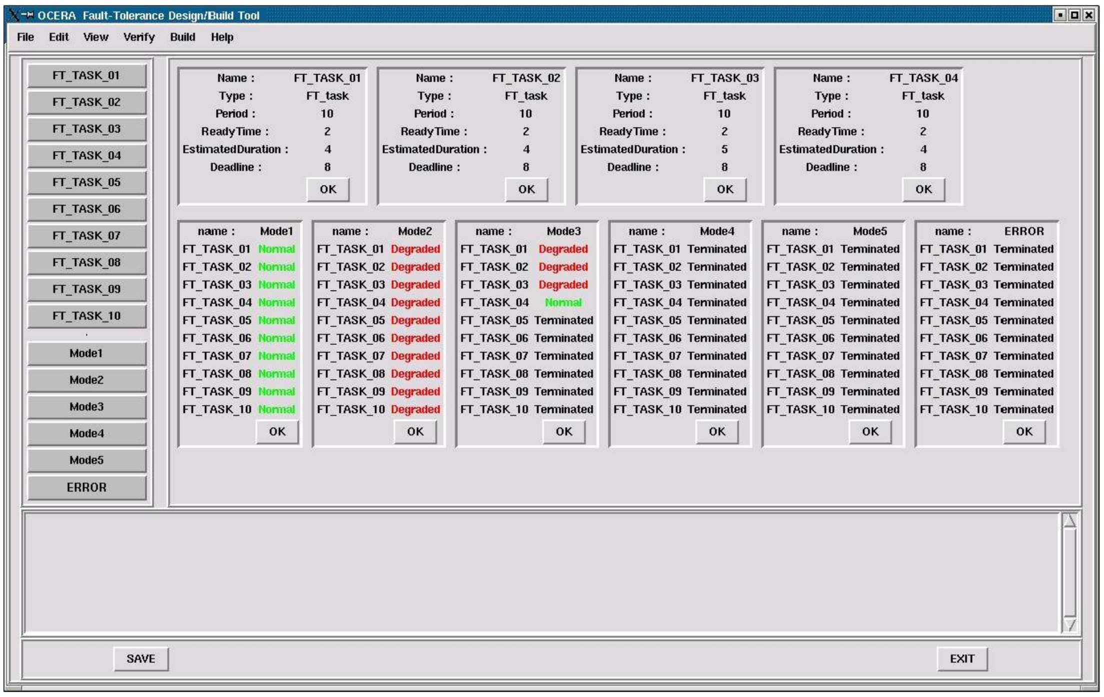
The second one must be done by hand by the application developer

The third step combines files issued by the two previous steps and links it with ft-components

The whole process is described in the following image :



The FT-Builder provides various facilities to define tasks, modes, transitions, to edit and view them. It permits also to generate application model files used for application compilation. The following figure shows a general overview of the tool where tasks and modes are displayed. In the next sections we review dedicated acquisition windows for task, modes, and transition specification.



Name :	FT_TASK_01	Name :	FT_TASK_02	Name :	FT_TASK_03	Name :	FT_TASK_04
Type :	FT_task	Type :	FT_task	Type :	FT_task	Type :	FT_task
Period :	10	Period :	10	Period :	10	Period :	10
ReadyTime :	2	ReadyTime :	2	ReadyTime :	2	ReadyTime :	2
EstimatedDuration :	4	EstimatedDuration :	4	EstimatedDuration :	5	EstimatedDuration :	4
Deadline :	8	Deadline :	8	Deadline :	8	Deadline :	8

name :	Mode1	name :	Mode2	name :	Mode3	name :	Mode4	name :	Mode5	name :	ERROR
FT_TASK_01	Normal	FT_TASK_01	Degraded	FT_TASK_01	Degraded	FT_TASK_01	Terminated	FT_TASK_01	Terminated	FT_TASK_01	Terminated
FT_TASK_02	Normal	FT_TASK_02	Degraded	FT_TASK_02	Degraded	FT_TASK_02	Terminated	FT_TASK_02	Terminated	FT_TASK_02	Terminated
FT_TASK_03	Normal	FT_TASK_03	Degraded	FT_TASK_03	Degraded	FT_TASK_03	Terminated	FT_TASK_03	Terminated	FT_TASK_03	Terminated
FT_TASK_04	Normal	FT_TASK_04	Degraded	FT_TASK_04	Normal	FT_TASK_04	Terminated	FT_TASK_04	Terminated	FT_TASK_04	Terminated
FT_TASK_05	Normal	FT_TASK_05	Degraded	FT_TASK_05	Terminated	FT_TASK_05	Terminated	FT_TASK_05	Terminated	FT_TASK_05	Terminated
FT_TASK_06	Normal	FT_TASK_06	Degraded	FT_TASK_06	Terminated	FT_TASK_06	Terminated	FT_TASK_06	Terminated	FT_TASK_06	Terminated
FT_TASK_07	Normal	FT_TASK_07	Degraded	FT_TASK_07	Terminated	FT_TASK_07	Terminated	FT_TASK_07	Terminated	FT_TASK_07	Terminated
FT_TASK_08	Normal	FT_TASK_08	Degraded	FT_TASK_08	Terminated	FT_TASK_08	Terminated	FT_TASK_08	Terminated	FT_TASK_08	Terminated
FT_TASK_09	Normal	FT_TASK_09	Degraded	FT_TASK_09	Terminated	FT_TASK_09	Terminated	FT_TASK_09	Terminated	FT_TASK_09	Terminated
FT_TASK_10	Normal	FT_TASK_10	Degraded	FT_TASK_10	Terminated	FT_TASK_10	Terminated	FT_TASK_10	Terminated	FT_TASK_10	Terminated

In this figure we can see an example of display showing tasks and modes .

Tasks and modes are listed in the left part ot the screen while details are displayed on the rigth part. The bottom part is devoted to the display of messages or to list entities such as tasks, modes and mode transitions (menu Edit/<entity>/List<entity>

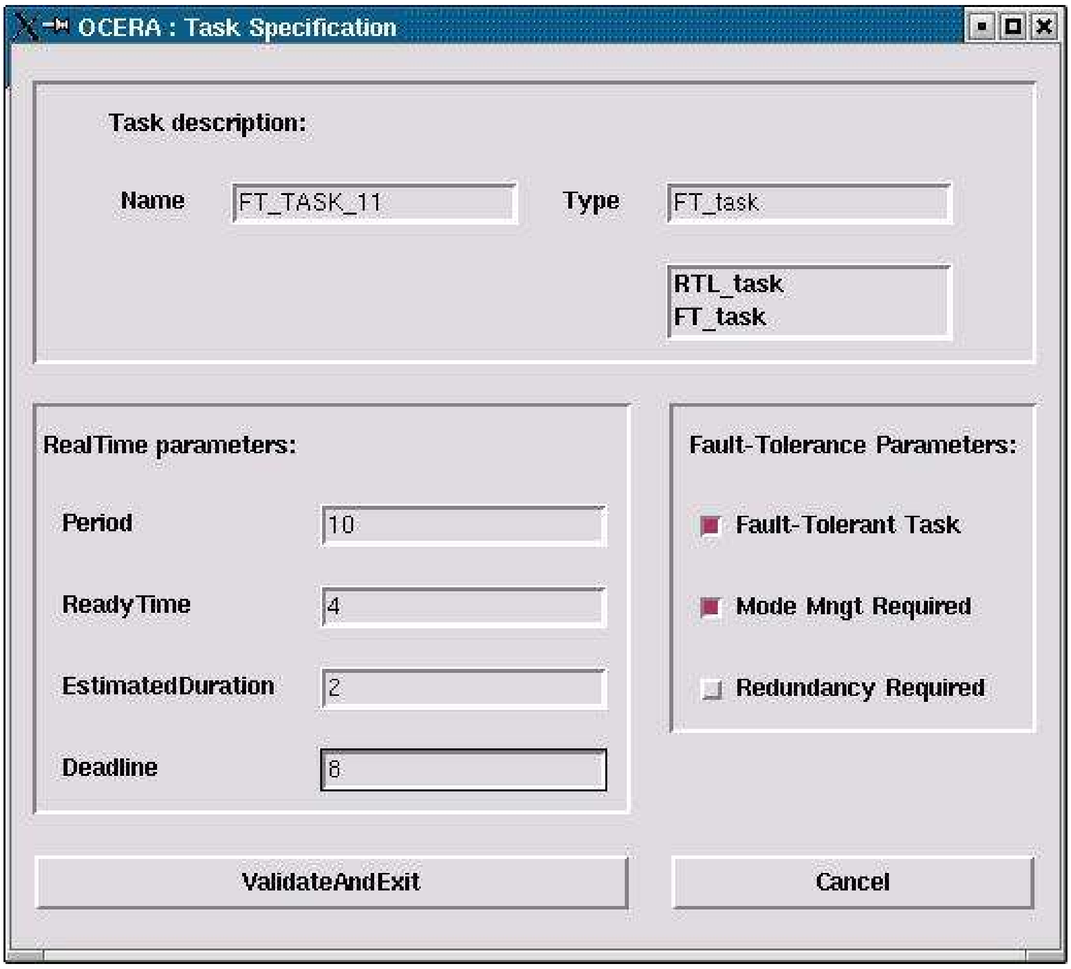
Real-time parameters are :

- Period,
- Ready Time,
- Estimated Duration
- Deadline

Modes are described by the list of tasks and related behavior that must be applied in the mode.

Task specification

The task specification consists in prividing ft_task real_time and ft_task parameters. This is done using the FT-Builder NewTask or ModifyTask facility.



OCERA : Task Specification

Task description:

Name: FT_TASK_11

Type: FT_task

RTL_task
FT_task

RealTime parameters:

Period: 10

ReadyTime: 4

EstimatedDuration: 2

Deadline: 8

Fault-Tolerance Parameters:

[✓] Fault-Tolerant Task

[✓] Mode Mngt Required

[ ] Redundancy Required

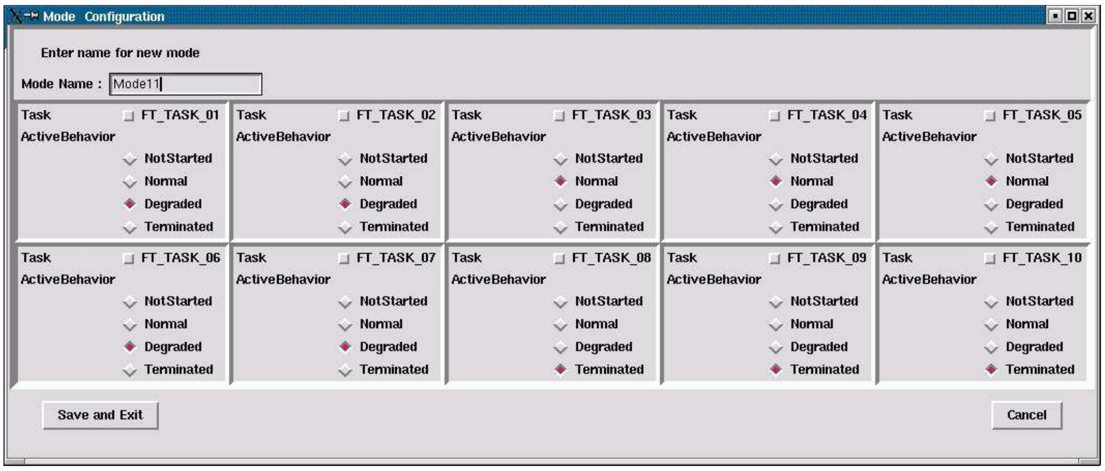
In the current (V1.0) implementation, only FT_tasks are handled and management of redundancy is not yet provided

Validation of consistency of real-time criteria is performed before storing information.

Mode specification

The mode specification consists in selecting for each task the behavior expected in the mode

This done using the Ftbuilder option NewMode facility


Mode Configuration

Enter name for new mode

Mode Name : Mode11

Task	ActiveBehavior
FT_TASK_01	NotStarted, Normal, [✓ Degraded], Terminated
FT_TASK_02	NotStarted, Normal, [✓ Degraded], Terminated
FT_TASK_03	NotStarted, [✓ Normal], Degraded, Terminated
FT_TASK_04	NotStarted, [✓ Normal], Degraded, Terminated
FT_TASK_05	NotStarted, [✓ Normal], Degraded, Terminated
FT_TASK_06	NotStarted, Normal, [✓ Degraded], Terminated
FT_TASK_07	NotStarted, Normal, [✓ Degraded], Terminated
FT_TASK_08	NotStarted, Normal, Degraded, [✓ Terminated]
FT_TASK_09	NotStarted, Normal, Degraded, [✓ Terminated]
FT_TASK_10	NotStarted, Normal, Degraded, [✓ Terminated]

The user has just to enter a mode name (which must be different from an existing one) and to select for each task, the right behavior to adopt in this mode.

Once these choices have been made, just save and exit (or cancel if you wish).

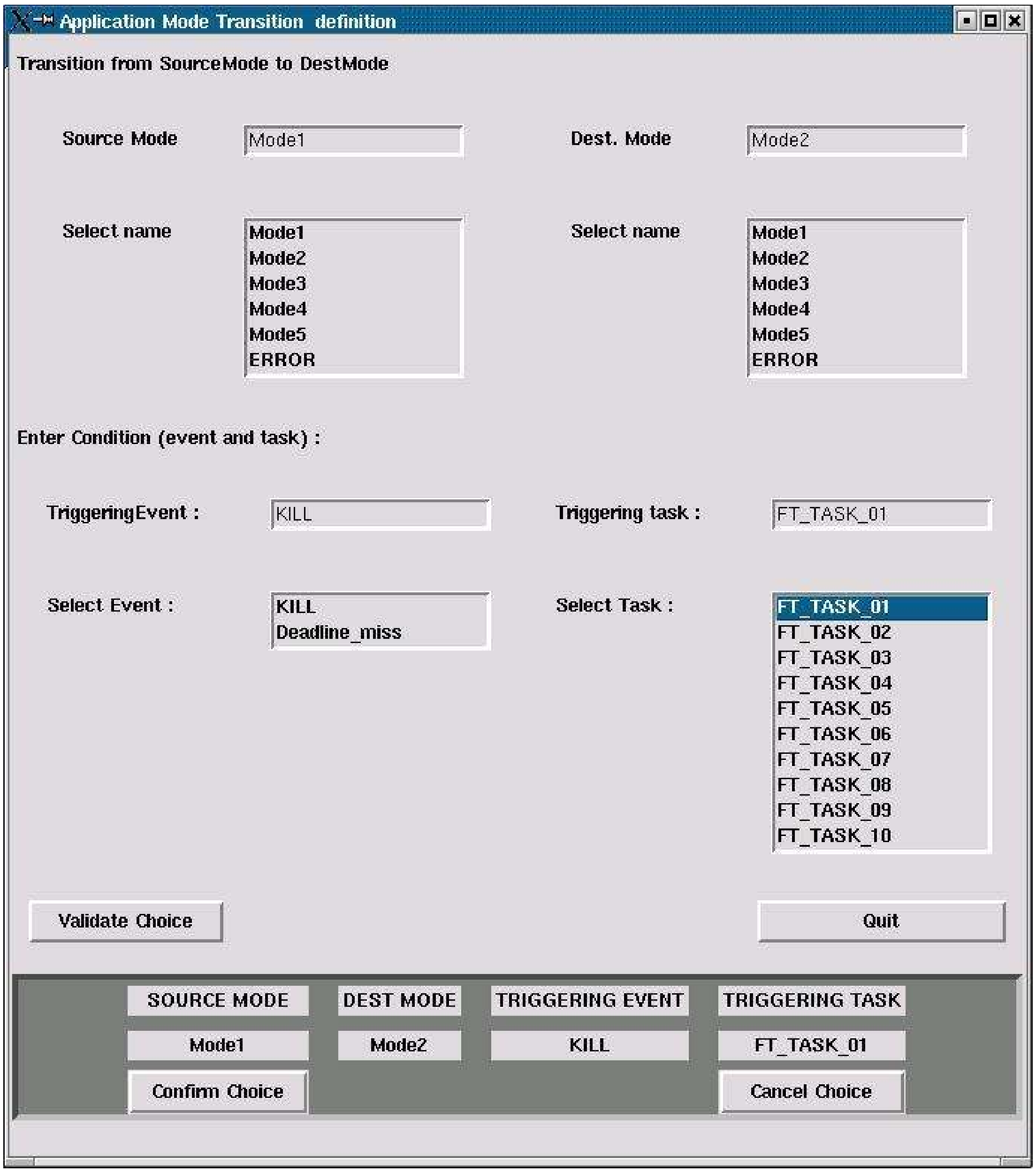
Mode transition specification

The mode transition specification consists in the specification of :

- a Source Mode
- a Destination Mode
- a transition condition

  where the transition condition consists of

  - a triggering event (the event can be KILL or deadline Miss)
  - a triggering task (the task that receive the event)


Application Mode Transition definition
Transition from SourceMode to DestMode
Select name
Select name
Enter Condition (event and task) :
Select Event :
Select Task :
Validate Choice
Quit
Confirm Choice
Cancel Choice

Source Mode: Mode1

Dest. Mode: Mode2

Source Mode select name: Mode1, Mode2, Mode3, Mode4, Mode5, ERROR

Dest. Mode select name: Mode1, Mode2, Mode3, Mode4, Mode5, ERROR

TriggeringEvent : KILL

Triggering task : FT_TASK_01

Select Event : KILL, Deadline_miss

Select Task : FT_TASK_01, FT_TASK_02, FT_TASK_03, FT_TASK_04, FT_TASK_05, FT_TASK_06, FT_TASK_07, FT_TASK_08, FT_TASK_09, FT_TASK_10

SOURCE MODE	DEST MODE	TRIGGERING EVENT	TRIGGERING TASK
Mode1	Mode2	KILL	FT_TASK_01

The user must enter the source and dest. Modes, then select the type of event and the triggering task.

Then he can validate his choice, a confirmation step displays a summary of the choices made for the transition, and waits for confirmation or cancellation.

The user can list the existing defined transitions with the Edit/Modes/ListModesTransitions facility.

The transitions are displayed in the bottom part of the main window.

4.3 API / Compatibility

Not applicable

Generates code to be used with FT (controller and appmon) components V2.0

4.4 Implementation issues

The implementation has been done in TCL/TK8.3 .

The current status of the tool permit the specification of application model including

- Tasks (name name and real_timeparameters)
- Modes
- Modes Transitions

As it is the tool can generate code necessary to instanciate internal databases of the two ft-components developed during the first phase of the project (ftappmon and ftcontroller).

This permit to support the Degraded Mode management facility.

The code code generated is currently limited to the production of files related to DataBases Information.

However information gathered could easily be used to provide an extension that would produce tasks code templates. This would insure consistency between the model and the code and facilitate maintainability of applications. One could easily change real_time parameters of taks and propagate change in the code. This has not been done yet.

An other limitation is the fact that tasks handled are limited to FT_tasks.

In a seconf phase of the project, Redundancy Management have been introduced, the tool will be extended in order to handle this new facility.

4.5 Tests and validation

4.5.1 Validation criteria

In a first stage validation criteria concern purely functional qualitative criteria.

4.5.4 Results and comments

The tool has been used to enter models related to the ft_controller example.

The tool is still in a construction step, and all expected facilities have not yet been implemented (mainly global verification facilities and additional faciilities to build the application), however, as it is it permits to enter completely information required for handling degraded mode management.

A first version has been tested by CTU which revealed a few errors, mainly due to not yet handled facilities.

These bugs have been corrected and a new version is now available.

4.6 Example

The **ftbuilder** contains examples that can be loaded using the File/OpenApplication facility of the menu.

You can define your own models.

One of the examples corerspond to the application example provided in the **ftcontroller**/examples directory (see section 3 for details).

4.6.1 How to run the example

Simply start the ftbuilder and use the options provided in the menu.

Once an application has been defined (tasks, modes and mode Transitions), the user can activate the Build Applicaion option which triggers the code generation process .

The generated files ***ft_appli_model.h*** and ***ft_appli_model.c*** are stored by default in **<ftbuilder_dir/appli_generated_files/<appli_name>/**.

They can then be copied into the application directory to be included during the compilation step.

4.7 Installation instructions

Just copy the ftbuilder directory in your favorite place, then you can start the ftbuilder by using the following command.

./FT_builder.tcl

The ftbuilder starts you can open an existing application and navigate through tasks, mods and modes transitions.

Creating a new application is quite simple.

The Help facility provides a summary of all options available.

Th information gathered by the tool is stored in an internal databases that is located under

<ftbuilder_dir/saved_appli_files/<appli_name>/.

The code generated by the BuildApplication option is stored in

<ftbuilder_dir/appli_generated_files/<appli_name>/.

You can change these locations by entering your modifications in the ***FT_preferences.tcl file.***